1 Sequencing and scheduling

In sequencing and scheduling problems we find a large variation of problems concerning the assignment and timing of tasks or jobs, to resources, often referred to as machines. Quite often jobs can be seen as (half-open) time intervals of specified lengths (often referred to as processing time, \( p_j \)), that have to be placed without overlap on half-lines \([0, \infty)\). In real life this resembles the requirements that a job can be handled by one person at a time, one person can only handle one job at a time, and handling of a job should not be interrupted.

If a job \( j \) is handled in interval \([C_j - p_j, C_j)\), then \( C_j \) is called its completion time. Often it is clear which resource is needed for a job. If there is a choice, let \( \mu(j) \) denote the resource or machine selected to process job \( j \). A schedule is then the assignment \((\mu_j, C_j)_j\).

1.1 Objectives

The value of a schedule depends on the objective of the problem. Usually we mean to minimize some expression depending on the values \( C_j \):

- \( C_{\text{max}} = \max_j C_j \) is called the makespan; it focuses on releasing the total set of resources as early as possible.
- \( L_{\text{max}} = \max_j L_j = \max_j C_j - d_j \) is called the lateness; it focuses on completing jobs in time. Here \( d_j \) is called the jobs due date. If \( L_{\text{max}} \leq 0 \), then all jobs are in time. If \( L_{\text{max}} < 0 \), you have created some slack time, and your schedule is more robust.
- \( \sum_j C_j \) is the total completion time. Minimizing this, or minimizing \( \frac{1}{n} \sum_j C_j \) (where \( n \) denotes the number of jobs) is good for the average job.
- Minimizing \( \sum_j w_j C_j \) is good for the average job, and even better for important jobs (with higher weight \( w_j \)).
- \( \sum_j U_j \) is the total number of late jobs. Here \( U_j = 1 \) if a job is late, i.e. \( C_j > d_j \), and \( U_j = 0 \) otherwise. In a situation in which there is more work than can be handled this may be a reasonable objective to work with.

1.2 Restrictions

Jobs may be restricted, in that for any schedule, it is required that a job starts not earlier than some prefixed time \( r_j \) (release date), or that it must be finished strictly not later than a certain time \( d_j \) (deadline). Note the subtle difference between deadlines and due dates (part of the objective).

Jobs may further be restricted by precedence constraints. These constraints can be denoted by \( J_j \rightarrow J_k \), meaning that job \( J_j \) has to have been completed before \( J_k \) can be processed. Hence \( C_j \leq C_k - p_k \).

More complex precedence constraints, referred to as communication delays, can hold if processing of job \( J_k \) requires the result of \( J_j \) which is immediately available if both jobs are processed on the same machine, but comes with a delay, if jobs are processed on different machines.

Machines can be restricted in that they may not always be available during the interval \([0, \infty)\).
Further we distinguish between different sets of machines that are available in parallel: \( Pm|\cdot| \) refers to problems in which \( m \) identical machines are available. Each of them can handle any job, requiring always time \( p_j \) for job \( j \). A job has to be handled by one of the machines. \( Qm|\cdot| \) refers to uniform machines, meaning that machine \( M_i \) has a speed \( s_i \), and his machine can handle job \( j \) with processing requirement \( p_j \) in a time interval of length \( p_j/s_i \). Finally we have unrelated machines, which is denoted by \( Rm|\cdot| \). Here machine \( M_i \) can handle job \( j \) requiring a time of \( p_{ij} \) (which may be \( \infty \)). In the above notation, \( m \) denotes the number of machines, a priori fixed. If the number of machine is not fixed, we leave out \( m \).

1.3 Relaxations

By allowing ourselves some flexibility, hard scheduling problems may be turned into easier ones. For jobs we often consider preemptive schedules. In the three field notation, the second field concerns job characteristics, and this property of the problem class under consideration is denoted as \( \cdot|pmtn|\cdot \) (preemption). It indicates that instead of one interval of prefixed length we need one or more non-overlapping intervals, with a total length of \( p_j \), for job \( j \). These intervals may be assigned to different resources, as long as they do not overlap. We say that a job is being interrupted and resumed.

A relaxation for machines may be that they can have capacity more than 1, that is that they can handle more than one job at a time. Usually we allow infinite capacity.

2 Claims and proofs

In sequencing and scheduling theory we usually describe algorithms to solve optimally or approximately certain types of problems. Often the quality of an algorithm is measured by analysis of the worst case relative deviation from the optimum. Considering a problem class \( \mathcal{I} \) (of minimization problems), and an algorithm \( A \), then the worst case ratio \( wcr(A) \) is defined as

\[
wcr(A) = \sup_{I \in \mathcal{I}} \frac{A(I)}{OPT(I)}
\]

where \( OPT(I) \) denotes the (unknown) optimal schedule value for problem \( I \), and \( A(I) \) denotes the value of the schedule for \( I \), obtained by applying algorithm \( A \) on instance \( I \). Evidently there may be algorithms with a bad worst case performance, but with a very good average performance. However, average performance analysis is (1) much harder and (2) dependent of the chosen probability distribution over the set of problem instances.

2.1 List scheduling

For \( Pm||C_{\text{max}} \) the algorithm LS (List Scheduling) is defined by

1. For machines \( i = 1, \ldots, m \) set current load to zero: \( \lambda(i) := 0 \);
2. For jobs \( j = 1, \ldots, n \), assign job to machine \( \mu(j) := \arg\max \lambda(i) \); set \( C_j = \lambda(\mu(j)) + p_j \); and reset \( \lambda(\mu(j)) := C_j \);
In words: we assign the jobs in the given order to the machine with the current minimum machine load.

Claim: wcr(\text{LS}) = 2 - \frac{1}{m}.

Proof: (a) Let \( k \) be the job in instance \( I \) for which \( C^\text{LS}_{\text{max}}(I) = C_k; \) then at time \( T := C_k - p_k \) all machines were busy, hence \( C^\text{OPT}_{\text{max}}(I) \geq T + p_k. \) Further we have that \( C^\text{OPT}_{\text{max}}(I) \geq p_k. \) It follows that \( C^\text{LS}_{\text{max}}(I) = T + p_k \leq (1 + \frac{m-1}{m})C^\text{OPT}_{\text{max}}(I); \) (b) consider the instance \( I_m \) with first \( m(m-1) \) jobs with processing time 1, and next one job of length \( m. \) Then \( C^\text{LS}_{\text{max}}(I_m) = 2m - 1, \) whereas \( C^\text{OPT}_{\text{max}}(I_m) = m. \)

It follows from the analysis that it is better to have a small job \( k \) at the end. Also intuitively one tends to schedule the big jobs first. To make it more precise, LPT (Longest Processing Time first) is defined as follows: list jobs in order of \( p_j \downarrow \), then apply LS on this sequence.

Remark: the claim that LPT(\( I \)) = OPT(\( I \)), for \( n_l \leq 2n_l, \) where \( n_l, m_l \) denote the number of jobs and machines, respectively, for instance \( I \), is false. Consider the instance on 3 machines, 6 jobs with processing times 6, 3, 3, 2, 2, 2. We have OPT = 6, whereas LPT = 7.

Claim: LPT(\( I \)) \leq \text{GREEDY}(\( I \)), for \( n_l \leq 2n_l, \) where \text{GREEDY}(\( I \)) is defined by sorting jobs from large to small, and then assigning jobs \( j \) to machine \( j \), for \( 1 \leq j \leq m, \) and to \( 2m + 1 - j, \) for \( m + 1 \leq j \leq 2m. \)

Proof: by construction, LPT assigns job \( j \) to machine \( j \) for \( j \leq m. \) For larger \( j, \) machines \( m, \ldots, 2m + 1 - j \) are considered, and at this point the load of machine \( 2m + 1 - j \) is \( p_{2m+1-j}. \) Hence the completion time \( C^\text{LPT}_j \leq p_{2m+1-j} + p_j, \) and therefore \( C^\text{OPT}_{\text{max}} \leq C^\text{GREEDY}_{\text{max}}. \)

Claim: LPT(\( I \)) = OPT(\( I \)), in case there is an optimal schedule with at most two jobs per machine.

Proof: for \( n_l \leq m_l \) the claim is obvious. For \( n_l > m_l, \) consider an optimal schedule with at most two jobs per machine. Then, without loss of generality, each machine processes at least one job. Further without loss of generality, the \( k \) machines processing a single job, process the \( k \) largest jobs. Finally we may assume that the remaining \( m_l - k \) machines process the remaining \( 2(m_l - k) \) jobs in GREEDY fashion. For, suppose that machine 1 processes jobs of length \( x_1, x_2, \) and machine 2 processes jobs with lengths \( y_1, y_2, \) with \( x_1 \) maximum and \( y_2 \) minimum. Then exchanging \( x_2 \) and \( y_2 \) yields machine loads \( x_1 + y_2 \) and \( y_1 + x_2, \) both not larger than \( x_1 + x_2. \) With OPT(\( I \)) \geq \text{GREEDY}(\( I \)) \geq LPT(\( I \)) the claim follows.

Claim: LPT(\( I \)) \leq (\frac{4}{9} - \frac{1}{3m}) \text{OPT}(\( I \)).

Proof: suppose the claim is false, and consider the instance \( I \) with smallest \( n_l + m_l \) contradicting the claim. By minimality of \( I, \) job \( n_l \) must be on critical machine (machine with load equal to makespan). By the previous claim, since LPT(\( I \)) > OPT(\( I \)), any optimal schedule has at least one machine processing three jobs. Hence it follows that any optimal schedule has makespan at least \( 3p_{n_l}, \) Hence with \( T \) denoting the start time of job \( n_l, \) LPT(\( I \))
\[ T + \frac{1}{m} p_{n_1} + \frac{m-1}{m} p_{n_1} \leq (1 + \frac{m-1}{3m}) \text{OPT}(I). \]

### 2.2 Minimizing the number of late jobs

The following algorithm due to Moore and Hodgson schedules jobs on a single machine minimizing the number of late jobs:

1. sort jobs in order of increasing due date: \( d_j \rceil \);  
2. start with scheduled job set \( J_0 = \emptyset \), load \( \lambda = 0 \);  
3. for \( j = 1, \ldots, n \), if \( \lambda + p_j \leq d_j \), then \( J_j = J_{j-1} \cup \{j\} \); \( \lambda = \lambda + p_j \); otherwise, let \( j_{\text{max}} \in J_{j-1} \cup \{j\} \) have largest processing time; set \( J_j = J_{j-1} \cup \{j\} \backslash \{j_{\text{max}}\} \); \( \lambda = \lambda + p_j - p_{j_{\text{max}}} \).  
4. Schedule jobs in \( J_n \) in order of due date; discard jobs not in \( J_n \) or schedule them in any order after the jobs in \( J_n \).

Claim: Moore-Hodgson yields a schedule with a minimum number of late jobs.

Proof: assume the jobs are already in due date order. Then the claim is equivalent to: (*) for each \( k = 1, \ldots, n \), \( J_k \) is a maximum cardinality feasible subset of \( S(k) := \{1, \ldots, k\} \). We prove a slightly stronger statement: (***) for each \( k = 1, \ldots, n \), \( J_k \) is a maximum cardinality feasible subset of \( S(k) := \{1, \ldots, k\} \), and among all maximum cardinality feasible sets, \( J_k \) has smallest total length.

To prove (***) let \( N(k) \) denote the true maximum cardinality of a feasible subset of \( S(k) \), and let \( F_k \) denote such a maximum cardinality set of minimum total length. Here feasible means that the subset can be scheduled in time, which can be tested by checking the EDD-schedule for \( F_k \). Assuming that (***) is not true, consider the smallest counter-example. Evidently, \( n > 1 \), since for a single job, \( J_1 = \emptyset \) if and only if \( N(1) = 0 \) if and only if \( p_1 > d_1 \).

By minimality we have that \( |J_{n-1}| = N(n-1) \). Note that \( |J_{n-1}| \leq |J_n| \leq |J_{n-1}| + 1 \), and similarly, \( N(n-1) \leq N(n) \leq N(n-1) + 1 \), and furthermore \( |J_n| \leq N(n) \). As (***) is not true we must have (a) \( |J_{n-1}| = |J_n| \) and \( N(n) = N(n-1) + 1 \), or we have (b) \( |J_n| = N(n) \) but \( J_n \) is not of minimum total length. If we are in case (a), then there is a maximum cardinality set \( F_n \) of size \( N(n) \) that must contain job \( n \). But then \( F_n \backslash \{n\} \) has size \( N(n-1) \) and has total length at least that of \( J_{n-1} \). \( F_n \) is feasible, hence the EDD-schedule is feasible. It ends with job \( n \), which means that \( J_{n-1} \cup \{n\} \) is also feasible, hence \( J_n = J_{n-1} \cup \{n\} \) contradicting (a).

If we are in case (b) and \( N(n) = N(n-1) + 1 \), then \( n \in J_n \), and \( n \in F_n \) with \( |F_n| = |J_n| \), and \( p(F_n) < p(J_n) \). But then \( p(F_n \backslash \{n\}) < p(J_{n-1}) \), contradicting the minimum length of \( J_{n-1} \). If we are in case (b) and \( N(n) = N(n-1) \), then \( n \in F_n \) with \( |F_n| = |J_n| \), and \( p(F_n) < p(J_n) \leq p(J_{n-1}) \). Let \( j_{\text{max}} = \arg \max \{p(j) \mid j \in J_{n-1} \cup \{n\}\} \), and let \( j_1 = \max \{j \mid j \in J_{n-1} \backslash F_n\} \). Then \( p(F_n \cup \{j_1\} \backslash \{n\}) < p(J_n) + p(j_1) - p(n) \leq p(J_n) + p(j_{\text{max}}) - p(n) = p(J_{n-1}) \). Note that if \( p(j_1) \leq p(n) + p(J_{n-1}) - p(F_n) \), then the set \( F_n \cup \{j_1\} \backslash \{n\} \) is feasible, contradicting the minimality of \( p(J_{n-1}) \). If \( p(j_1) \) is larger we have \( p(j_1) > p(n) + p(J_{n-1}) - p(F_n) > p(n) + p(J_{n-1}) - p(J_n) = p(j_{\text{max}}) \), a contradiction.

### 2.3 Maximizing slot assignment

The following problem is a special case of minimizing number of late jobs: consider a (communication) processor that, during a given time slot, can process a number of packets, up
to a certain total size $D$. In this special case packet sizes are not arbitrary but they can only have values from $\{a_1, \ldots, a_K\} \subset \mathbb{N}$, with the additional property of divisibility: $a_i|a_{i+1}$, for all $i < K$. In the UMTS applications these packets are being transmitted and therefore each packet should be processed in time; each packet $j$ has a due date $d_j \in \{1, \ldots, d\}$. If a packet is not processed it gets lost and some back-up action has to be taken. Therefore we want to assign as many packets to a feasible time slot as possible. A feasible assignment $\sigma$ is a mapping $\sigma : J \rightarrow \{0, \ldots, d\}$, with $\sigma(j) \leq d_j$ and satisfying $\sum_{j: \sigma(j)=t} s_j \leq D$, for each $t \in \{1, \ldots, d\}$. Here $s_j$ denotes the size of $j$, and $\sigma(j) = 0$ means that job $j$ is not assigned a time slot.

To test whether a subset $S \subseteq J$ can be completely assigned can be done by applying the following dispatch rule, called LFD (Latest Fit Decreasing): sort jobs (packets) from large to small size, and assign each job, in this order, to the latest time slot with enough residual capacity. This procedure takes $O(dK)$ time.

Claim: if $S$ is feasible, then the LFD-rule will actually find a feasible assignment.

Proof: consider a feasible assignment $\sigma$ for $S$ and consider the jobs in dispatch order. Let $j$ be the first job in dispatch order that is scheduled in an earlier slot $\sigma(j)$ than the slot $\delta(j)$ suggested by the dispatch rule. Then the current residual capacity of slot $\delta(j)$ is at least $s_j$. All jobs $k$ in $S$, with $k > j$, and with $\sigma(k) = \delta(j)$, have size $s_k \leq s_j$, hence $s_k|s_j$. Now, either the remaining capacity of slot $\delta(j)$ is used by $\sigma$ for a total weight less than $s_j$, or it is used by a set of jobs, for which there is a subset with total weight exactly $s_j$. Here we use the divisibility of the sizes of packets. In either case, we can move job $j$ from time slot $\sigma(j)$ to slot $\delta(j)$, as $d_j \leq \delta(j)$, and move a subset of smaller jobs with total size $\leq s_j$ from time slot $\delta(j)$ to $\sigma(j)$, which is earlier. Repeating this procedure, any schedule $\sigma(S)$ can be turned into a feasible schedule that is the result of the above dispatch rule.

To find a maximum cardinality assignment of time slots to packets we can use the following algorithm, called Bottom-Up, that takes only $O(dK)$ time as well. Actually, the procedure solves a slightly more general problem, which we first define as follows:

Generic-Slot-Assignment: We are given $K$ pairwise divisible sizes $a_i$, $a_i|a_{i+1}$, $d$ time slots $t$, and numbers $N(i,t)$ of jobs of size $a_i$ and due date $t$. Further we are given a processor with capacity $D(t)$ in time slot $t$, with $a_1|D(t)$, for all $t$. Let $X$ denote a subset of jobs of size $a_1$, with the property that there exists a feasible assignment for $X$. Find a slot assignment of as many packets as possible, including all packets from $X$.

Bottom-Up: for weight class $a_1$, test if $S_1$, the set of all jobs with size $s_j = a_1$, can be assigned (by applying the LFD-rule).

If this is not possible, schedule jobs in $X$ by LFD, and assign the remaining jobs from $S_1$ in order of increasing due date as early as possible, and as long as possible. There will be a slot $t$ such that capacities $D(1), \ldots, D(t)$ can be fully used for all jobs in $X \cap \{j|d_j \leq t\}$, and some (earliest) jobs from $(S_1 \setminus X) \cap \{j|d_j \leq t\}$, and such that all jobs in $S_1 \cap \{j|d_j > t\}$ can be scheduled in time slots above $t$. Apply Bottom-Up inductively on the remaining set of jobs with $d_j > t$.

If $S_1$ can be scheduled in total, define capacities $B(t) = D(t) \mod a_1$, for each time slot $t$,
and sort the jobs in $S_1$ in increasing due date order. Schedule as many jobs as possible from this list, using capacity $B(t)$. Group the remaining jobs consecutively in chunks of $\frac{a_2}{a_1}$ jobs each. Each chunk is assigned as due date the due date of its first element. Consider these groups as packets of size $a_2$, naming this set $X'$ and add them to the set of jobs of weight $a_2$. Apply recursively Bottom-Up on the instance with $K - 1$ sizes $a_2, \ldots, a_K$, $d$ time slots, processor capacities $D'(t) = D(t) - B(t)$, and special job set $X'$.

Claim: Bottom-Up finds a maximum cardinality assignment for $J$, and this assignment does schedule all jobs in $X$, and respects processor capacities $D(t)$.

Proof: follows from an inductive analysis of an optimal solution $\sigma$. The induction basis is the observation that it is true in case there is only one weight class or zero time slots.

Next, let $M(i)$ denote the number of jobs of weight $a_i$, scheduled by Bottom-Up. First, without loss of generality, an optimal solution $\sigma$ exists, which assigns at least the $M(1)$ jobs of weight $a_1$ selected by Bottom-Up.

Further, no schedule exists that assigns more than $M(1)$ packets, as Bottom-Up schedules either all jobs in $S_1$, or $\sum_{\tau \leq t} D(\tau)/a_1$ jobs from $S_1 \cap \{j | d_j \leq t\}$ and all jobs from $S_1 \cap \{j | d_j > t\}$.

Let us assume that all $S_1$-jobs fit. As capacities $B(t)$ can only be used by packets of size $a_1$, we may assume without loss of generality, that any optimal schedule $\sigma$ makes optimal use of this capacity filling them with $S_1$-jobs, giving priority to lower due date jobs. The remaining slots occupied by $S_1$-jobs are, without loss of generality, used by lower due date packets as much and as early as possible. As the remaining capacities $D(t) - B(t)$ are multiples of size $a_2$, the $S_1$-jobs occupy these capacities in groups of cardinality $\frac{a_2}{a_1}$ each. By construction, as we know that all $S_1$-jobs fit, we know that a schedule for the $X'$-jobs exists. We can apply induction and know that the remaining jobs are optimally scheduled.

Now assume that not all $S_1$-jobs fit. Then the slots up to $t$ are maximally filled, and for the higher slots we apply induction directly, replacing $S_1$ by $S_1 \cap \{j | d_j > t\}$. Note that there exists a schedule for $S_1 \cap \{j | d_j > t\}$.

The original problem is now solved by setting $D(t) = D - (D \mod a_1)$, for all $t$, and $X = \emptyset$, applying Bottom-Up for this Generalized-Slot-Assignment problem.