Applications of Yao's Minimax Principle

Game Tree evaluation

We study the problem of evaluating a game tree, which is a rooted tree of the type $T_{d,k}$, where the root and every internal node (i.e. all nodes except the leaves) have d children and all the leaves of the tree are at distance $2k$ from the root. Every node corresponds to a Boolean expression. The root and every node at even distance from the root corresponds to an AND-type Boolean expression, which is assigned the value 1 (corresponding to TRUE) if and only if all of its children have value 1. All nodes at odd distance from the root correspond to OR-type Boolean expressions, which get the value 0 if and only if all of its children have value 0. An instance of the problem is given by a specification of 0-1 values for all the leaves of the tree and the aim is to determine the value of the root.

We will be interested in the number of leaves that we need to read to conclude the correct value of the root. As in the book we will concentrate on the game tree of the type $T_{2,k}$, i.e., $d = 2$, a binary game tree, each node of which, except the leaves, has two children. Such a tree has $2^{2k} = 4^k$ leaves.
Make for yourself Problem 2.1 to see that if an algorithms player specifies a deterministic way in which the order in which the leaves will be read is determined, then the instance player, acting as an adversary, can provide an instance such that the first player will be enforced to read all the leaves.

A randomised algorithm may do considerably better, since the adversary (the input player) would not be able to hide the 0-child of an AND-node and the 1-child of an OR-node. The randomised algorithm which is in fact best possible is the straightforward one. In evaluating a node it chooses one of the children first with probability 1/2. This strategy is applied recursively to the child chosen. So the algorithm is specified top-down.

We prove that this algorithm has to read an expected number of leaves equal to 3^k. It is useful to realise that if an AND-node is to get value 1 then, unavoidably, both its 2 OR-children will have to be evaluated to get value 1. But for an OR-gate to get value 1 one of its children can already give the certificate for value 1, and this one will be picked with probably 1/2 by our algorithm. Similarly, if an OR-node has value 0 then both its AND-children must be evaluated to be 0, but for each of them one child is enough to certify the value 0.

The proof that the RA has to read an expected 3^k number of leaves is by induction. The basis $k = 1$ is quite easy to prove by the two observations just given. Suppose now that it is true that evaluating any $T_{2,k-1}$ tree requires the reading of an expected number of 3^{k-1} leaves. Consider now a (sub)tree having an OR-node as its root, each of its two children being the root of a $T_{2,k-1}$ tree. If this OR-node would have value 1, then at least one of its children returns 1,

and this child is chosen with probability 1/2. This creates an expected number of expected leaves to be read bounded by

$$\frac{1}{2} \cdot 3^{k-1} + \frac{1}{2} \cdot 2 \cdot 3^{k-1} = \frac{3}{2} \cdot 3^{k-1}.$$

If the OR-node would have value 0 it would always cost an expected number of leaves to be read bounded by

$$2 \cdot 3^{k-1}.$$

Now consider the root of $T_{2,k}$. If it has value 1 then both its OR-children has value 1 costing an expected number of leaves to be read bounded by

$$2 \cdot \frac{3}{2} \cdot 3^{k-1} = 3 \cdot 3^{k-1} = 3^k.$$

If the AND-root would evaluate to 0. Then with probability 1/2 only one of its children has to be evaluated to be 0. With probability 1/2 the first child chosen would return 1 and then the second child will have to be evaluated as well (returning 0). This amounts to an expected number of leaves to be read bounded by

$$\frac{1}{2} \cdot 2 \cdot 3^{k-1} + \frac{1}{2} \left(\frac{3}{2} \cdot 3^{k-1} + 2 \cdot 3^{k-1} \right) = 1 \cdot 3^{k-1} + (1\frac{3}{4}) \cdot 3^{k-1} < 3^k.$$

Now we will use Yao's Minimax Principle to find out how good this 3^k is. I will disappoint you in advance for not showing you the lower bound of 3^k for the expected number of leaes to be read by any randomised algorithm. It is actually the best lower bound, but the analysis is far too intricate for educational purposes as an example of the application of Yao's Minimax Principle.

What will we show? Let us denote the total number of leaves by n: i.e., $n = 4^k$. Then $3^k \approx n^{0.793}$. As in the book I will show a lower bound of $n^{0.694}$.

According to Yao's Minimax Principle we are to give a random instance and analyse the expected number of leaves that any deterministic algorithm will have to read. We choose the random instance obtained by giving each leave value 1 independently with probability $p = (3 - \sqrt{5})/2$. For the analysis of this random game tree we replace all AND- and OR-nodes by NOR-nodes. A NOR-node has value 1 if and only if both its children have value 0.
Make Exercise 2.2 for yourself to check that indeed the value of a $T_{2,k}$ is equal to the value of a binary tree all of whose nodes are NOR-nodes.

The probability that a NOR-node whose two children are leaves gets value 1 is the probability that both leaves have value 0, which is

$$\left(1 - \frac{3 - \sqrt{5}}{2}\right)^2 = \left(\frac{\sqrt{5} - 1}{2}\right)^2 = \frac{3 - \sqrt{5}}{2} = p.$$

But this implies that every NOR-node in the tree has value 1 with probability p, and independently from all other nodes at the same level. We use the following theorem (Proposition 2.7 in the book), which is also intuitively correct.

Proposition 2.7 in [MR]. *Let T be a NOR-tree each of whose leaf-values are set equal to 1 with probability $\bar{p}$, independently, for some fixed $\bar{p}$. Let $W(T)$ be the minimum expected number of leaves to be read for evaluating T over all deterministic algorithms. Then there exists a depth-first pruning algorithm having $W(T)$ as the expected number of leaves to be read.*

A depth-first pruning algorithm is an algorithm that for every node, starting at the root, tries to evaluate one if its children first, before doing anything else. It may happen that the child has value 1 in which case the other child does not need to be considered anymore; the other child, i.e., the subtree having this child as root is then pruned.

Therefore we restrict our attention to depth-first pruning algorithms. We have already established that in our random instance every node gets value 1 with probability $p = (3 - \sqrt{5})/2$. Let $W(h)$ denote the expected number of leaves to be read for evaluating a node at distance h from the leaves. Then

$$W(h) = W(h-1) + (1-p)W(h-1).$$

Viz., one of the two children has to be evaluated for sure and with probability $(1-p)$ it gets value 0, in which case also the second child needs to be evaluated. Using that h, the distance of the root from the leaves is $h = \log_2 n$ leads in the recursion to $W(h) = n^{0.694}$.

On-line scheduling

The following application of Yao's Minimax Principle comes from on-line algorithms. This is the subject of Chapter 13 of [MR]. I advise you to read Sections 13.1 and 13.3 (and the bit of 13.2 that introduces the notion of oblivious adversary). It concerns an example of an on-line problem in which information is revealed to the algorithm one-by-one, and an irrevocable action is to be taken. I will concentrate on an on-line problem in which information arrives over time instead of one-by-one.

Now we consider an on-line model in which time plays a role. Before any time t nothing is known what comes at time t or after. Decisions may be postponed at the price of time that is passing by, for which one pays in objective value. Consider the off-line scheduling problem $1 \mid r_j \mid \sum C_j$:

Total completion time scheduling with release dates
Given a single machine and n jobs, each having a processing time p_j and a

release date r_j. Find a schedule of the jobs such that each job j is scheduled in an uninterrupted time interval of length p_j which does not start before r_j and such that the sum of the completion times of the jobs is minimised.

What is known on the computational complexity of the off-line problem? This problem is NP-hard, but there exists a PTAS. The version without release dates is easily seen to be solved by the *shortest processing time first* rule (SPT). (Just use a simple exchange argument.) Similarly easy it is to prove that the version in which preemption of the jobs is allowed is solved by the *shortest remaining processing time first* rule (SRPT).

We are interested in the online version (and we are not interested in running times). We assume that at time 0 we know nothing. Jobs are released over time and as soon as a job j is released we immediately get to learn its processing time. Before the release time we know nothing about the job, not even that it exists. Once we have a job and the machine is idle we may decide to start the job, and then we have to finish it uninterruptedly, or we may wait and see if other (smaller) jobs arrive soon, which we may like to give precedence. So we are allowed to postpone decisions. Only history is irrevocable.

Let us first think about how things can go wrong for a *deterministic* online algorithm due to this lack of information: i.e, let us try to find a lower bound on the *competitive* ratio for any algorithm.

Let $Z^*(I)$ be the value of an optimal solution of instance I and let $Z^A(I)$ be the value produced by an online algorithm A on instance I. The competitive ratio of online algorithm A is

$$\max_{I \in \mathcal{I}} \frac{Z^A(I)}{Z^*(I)}.$$

A lower bound on the competitive ratio of any online algorithm is given by

$$\min_{A \in \mathcal{A}} \frac{Z^A(I)}{Z^*(I)} \text{ for any } I \in \mathcal{I}$$

.

So let us play a malicious adversary who constructs a sick online instance I, depending on which action is taken by a deterministic online algorithm. Suppose at time 0 we see job 1 with $p_1 = 1$. Suppose an online algorithm starts this job at time x. Then if $x \geq 1$, the adversary will not release any further job and he finishes at time 1, while the online algorithm does not finish before time 2, yielding a competitive ratio of at least 2.

So suppose that $x < 1$. Then the adversary releases at time $x + \epsilon$ an enormous set of n jobs with processing time very small. Think of these jobs as having

processing time 0. Hence the adversary schedules the small jobs first adding $n(x+\epsilon)$ to the objective value and then the first job adding $x+\epsilon+1$; a total of $(n+1)(x+\epsilon)+1$. Whereas the online algorithm has just started job 1 and has to wait until it completes before it can schedule the small jobs, yielding a total completion time of $(n+1)(x+1)$. For $\epsilon \to 0$ and $n \to \infty$, we obtain a ratio of $(x+1)/x$, which is minimised by $x = 1$, and in that case is equal to 2.

Theorem 0.1. *No deterministic online algorithm for on-line minimising total completion time on a single machine can have a competitive ratio smaller than 2.*

In the literature several algorithms have been proposed that have a competitive ratio of 2. Let us study the most elegant among them. It is also the algorithm that can be used as the basis for a more sophisticated randomised algorithm, to which we come back later.

The algorithm just ignores the forbidding of preemption and uses the (online) SRPT-rule to build a schedule. This preemptive schedule is then used as the basis for building a non-preemptive schedule in the following way.

Algorithm: α-POINT.
Simulate on a single machine the (on-line by SRPT) construction of a preemptive schedule P. As soon as in this preemptive schedule an α-fraction of the processing time of a job has been completed, make it available for scheduling in the non-preemptive schedule by adding this job at the back of the list of jobs that became available before. Always schedule a job as soon as the machine completed all the previous jobs in the list of available jobs.

So just to repeat: as soon as a job in the preemptive schedule has seen an α-fraction of its processing time completed, it is made available for the non-preemptive schedule and is scheduled directly after the jobs for which an α-fraction of their processing time has been completed in the preemptive schedule before.

Our algorithm will not start the j-th job in the list of available jobs until the $j-1$-st is completed even if it may be profitable to switch them if at some time t they are both available and job $j-1$ has longer processing time than job j. Our analysis does not require this algorithmic enhancement.

Let C_j^α be the completion time according to the α-point schedule and C_j^P according to the preemptive schedule.

Theorem 0.2. *Given a preemptive schedule P for $1 \mid r_j, pmtn \mid \sum C_j$, the α-POINT algorithm yields in $O(n)$ time, a non-preemptive schedule in which, $\sum_j C_j^\alpha \leq (1+(1/\alpha)) \sum_j C_j^P$.*

Proof. # ..... THIS TIME DURING CLASS I SKIPPED THE PROOF ..... #

Index the jobs by the order of their α-points in the preemptive schedule P. We distinguish two situations.

In the first one, when j's α-point is reached (when job j becomes available for the non-preemptive online schedule), the machine on which we are building the online (non-preemptive) schedule is idle. In that case we start the job immediately and finish latest αp_j after C_j^P. Hence $C_j^\alpha \leq (1+\alpha)C_j^P \leq (1+(1/\alpha))C_j^P$, since $\alpha \leq 1$.

In the second situation at j's α-point the machine on which we are building the online (non-preemptive) schedule is processing some earlier available job. But then the last time the machine was idle was at the α-point of some earlier available job. Call this time t. And hence,

$$C_j^\alpha \leq t + \sum_{k=1}^{j} p_k. \tag{1}$$

Clearly $C_j^P \geq t$. We also know that $C_j^P \geq \alpha \sum_{k=1}^{j} p_k$, since the α-fractions of jobs $1, \ldots, j$ must run before time C_j^P. Plugging these last two inequalities into (1) yields the proof. □

Clearly, the best value for α in the range is $\alpha = 1$, yielding a ratio of 2.

The lower bound example we had before shows that for any α the adversary will at time 0 release a job with processing time 1 and at time $\alpha + \epsilon$ a set of n jobs of processing time 0. leading to a ratio $(\alpha+1)/\alpha$.

This suggest that we can fool the adversary by not revealing the α we choose. This is exactly the motivation of the randomized algorithm, which plays against an oblivious adversary. Such an adversary knows the distribution with which the randomized α-point algorithm is playing but does not see the realization. We wish to study the ratio: $E[\sum C_j^\alpha / \sum C_j^{OPT}]$.

Algorithm RANDOMIZED α-POINT: draw $\alpha \in (0,1]$ according to probability density function f and apply α-POINT.

..... THE ANALYSIS OF THE RANDOMIZED ALGORITHM I SKIPPED. IF YOU LIKE YOU MAY READ BY YOURSELVES THE LECTURE NOTES BELOW OR THE PAPER BY [CHEKURI ET AL. 2001] (SEE FOR THE REFERENCE BELOW). I RESUME THE LECTURE NOTES AT DERIVING THE LOWER BOUND. .....
To analyze the competitive ratio of this algorithm, let us do a slightly more precise analysis of what happens with the deterministic α-point algorithm. Take the completion time of job i in the preemptive schedule C_i^P. Let J_i^β be the set of jobs that by time C_i^P have processed exactly a β-fraction of their processing time. Let $S_i^\beta = \sum_{j \in J_i^\beta} p_j$. Let T_i be the total idle time before C_i^P. It is easy to see that

Lemma 0.3. $C_i^P = T_i + \sum_{0 \le \beta \le 1} \beta S_i^\beta$

Let us split it out a bit more:

$$C_i^P = T_i + \sum_{\beta < \alpha} \beta S_i^\beta + \sum_{\beta \ge \alpha} \alpha S_i^\beta + \sum_{\beta \ge \alpha} (\beta - \alpha) S_i^\beta$$

So, for any job j that had a $\beta > \alpha$-fraction of its processing time scheduled before C_i^P, $(\beta - \alpha)p_j$ is the total amount of processing time scheduled between its α-point and C_i^P. Let J^B be the set of jobs that have already passed their α-point by time C_i^P: $J^B = \{j \in \cup_{\beta \ge \alpha} J_i^\beta\}$. Since the α point of job i is certainly before C_i^P, the jobs that run before job i in the α-point schedule belong to J^B.

Now we are changing the preemptive schedule. Take the first job j in the preemptive schedule that reaches its α-point. Let this job have a fraction x_j of its processing time scheduled at C_i^P. Hence, as argued before, it has $(x_j - \alpha)p_j$ processing time scheduled between its α-point and C_i^P. This does not need to be a consecutive block of time in the preemptive schedule, but instead may consist of several pieces. We take all these pieces together and put them on the machine right after j's α-point. This may cause processing of parts of other jobs to be pushed later in time in the schedule, but the total amount of processing time done before C_i^P remains the same. For job j we have now an uninterrupted amount of $(x_j - \alpha)p_j$ processing time starting at its α-point. We repeat this for all other jobs in J^B.

Again I emphasize that by this operation only the parts of jobs in excess of their α-fraction have been grouped together in uninterrupted blocks of time and C_i^P has not become larger; we have only reshuffled pieces of jobs before C_i^P and did not insert idle time. Thus we have that for any job $j \in J^B$ a piece of size $(x_j - \alpha)p_j$ is scheduled without preemption after j's α-point.

Now we create a non-preemptive schedule by moving for each job in J^B all the other $p_j - (x_j - \alpha)p_j$ processing time (that is the processing time of j remaining to be done at time C_i^P plus the αp_j time done before j's α-point) directly after the piece of size $(x_j - \alpha)p_j$.

We then have a schedule in which all jobs that come before i in the α-point schedule are completed non-preemptively before job i is completed. The completion time of job i in this schedule is therefore an upper bound on its completion time in the α-point schedule. It has become bounded by:

$$C_i^\alpha \le C_i^P + \sum_{j \in J^B} p_j - (x_j - \alpha)p_j = C_i^P + \sum_{\beta \ge \alpha} (1 - \beta + \alpha) S_i^\beta$$

Inserting $C_i^P = T_i + \sum_{0 \le \beta \le 1} \beta S_i^\beta$ from the lemma yields

$$C_i^\alpha \le T_i + \sum_{\beta \ge \alpha} (1 + \alpha) S_i^\beta + \sum_{\beta < \alpha} \beta S_i^\beta. \tag{2}$$

Based on this bound we will be able to give a bound on the expected completion time for each job i in the randomised α-point schedule.

Theorem 0.4. *Given that α is drawn according to density f over $[0,1]$, we have for each job i that $E[C_i^\alpha] \leq (1+\delta)C_i^P$, with*

$$\delta = \max_{0\leq\beta\leq 1} \int_0^\beta \frac{1+\alpha-\beta}{\beta} f(\alpha)d\alpha.$$

The proof is really just taking (2) and taking the integral over α of the right-hand side times $f(\alpha)$

$$E[C_i^\alpha] \leq \int_0^1 \left(T_i + \sum_{\beta\geq\alpha} (1+\alpha)S_i^\beta + \sum_{\beta<\alpha} \beta S_i^\beta \right) f(\alpha)d\alpha.$$

and then taking outside of the integral the terms that are independent of α; i.e., T_i and S_i^β. You can read it yourself in the paper [Chekuri et al. 2001] (see for the reference below).

If we draw α according to density $f(\alpha) = \frac{e^\alpha}{e-1}$ then $\delta = \frac{1}{e-1}$. Thus, the following competitive ratio follows directly.

. During classes I resumed my lecture notes again from here on till the end .....#

Theorem 0.5. Randomised α-point *with α according to density $f(\alpha) = \frac{e^\alpha}{e-1}$ has competitive ratio $\frac{e}{e-1} \approx 1.58$.*

It can be shown that this is in fact the best ratio that can be obtained by any randomised algorithm on this on-line problem by applying Yao's minimax principle. We specify a random instance of the problem and analyse what any algorithm could attain in expectation on this instance.

- At time 0 one job with processing time 1 arrives.
- With probability $1 - \frac{e-1}{n}$ no further jobs arrive.
- With probability $\frac{e-1}{n}$ one set of $n-1$ jobs with processing time 0 arrives at some time x, which is a random variable over the interval (0,1] having probability density function $f(x) = \frac{e}{e-1}\ e^{-x}$.

First we derive the expected optimal objective value $E[\sum C_j^{OPT}]$ on the random instance. Observe that if only the first job is released the optimal value is equal to 1. If at any time $x \leq 1 - \frac{1}{n}$ the set of $n-1$ jobs is also released then it is profitable to schedule the $n-1$ jobs before the first job, yielding an objective value of $nx+1$. If the $n-1$ jobs arrive between time $1-\frac{1}{n}$ and 1 it is better to process the first job first giving a sum of completion times equal to n. These

observations lead to the following upper bound on the expected optimal solution value.

$$\begin{aligned} E[\sum C_j^{OPT}] &\leq \left(1-\frac{e-1}{n}\right)\cdot 1 + \frac{e-1}{n}\int_0^{1-\frac{1}{n}}(nx+1)f(x)dx \\ &\quad + \frac{e-1}{n}\int_{1-\frac{1}{n}}^{1} nf(x)dx \\ &\leq \left(1-\frac{e-1}{n}\right)\cdot 1 + \frac{e-1}{n}\int_0^{1}(nx+1)f(x)dx \\ &= 1-\frac{e-1}{n}+\frac{e}{n}\int_0^1 e^{-x}(nx+1)dx \\ &= 1-\frac{e-1}{n}+\frac{e}{n}(-ne^{-1}-ne^{-1}-e^{-1}+n+1) \\ &= e-1. \end{aligned}$$

Any deterministic on-line algorithm will have to start scheduling the first job at some point in time. We will denote the expected solution value of an on-line algorithm that does not start the first job before time t by $E[\sum C_j^{OL(t)}]$. Consider such an algorithm that starts processing the first job at time t. Obviously, if the set of $n-1$ jobs does not arrive the only job will be completed at time $t+1$. Otherwise, if the set of $n-1$ jobs arrives before t then it is obviously better for the algorithm to schedule the $n-1$ jobs first, before starting the (big) first job. Moreover, since the deterministic algorithm knows the distribution, he will start the first job immediately after having processed the other jobs since no further jobs will arrive. In this case a cost of $nx+1$ will be incurred. In case the set of $n-1$ jobs arrives after t then these jobs have to wait until the first

job is finished producing an objective value of $(t+1)n$. I.e.,

$$\begin{aligned}
E[\sum C_j^{OL(t)}] &= (1-\frac{e-1}{n})(t+1) + \frac{e-1}{n}\int_0^t \frac{e}{e-1}e^{-x}(nx+1)dx \\
&\quad + \frac{e-1}{n}\int_t^1 \frac{e}{e-1}e^{-x}(t+1)ndx \\
&= (1-\frac{e-1}{n})(t+1) + \frac{e}{n}(\int_0^t e^{-x}(nx+1)dx + \int_t^1 e^{-x}(nt+n)dx) \\
&= (1-\frac{e-1}{n})(t+1) + \frac{e}{n}(|_0^t(-nxe^{-x} - ne^{-x} - e^{-x}) \\
&\quad + |_t^1(-nte^{-x} - ne^{-x})) \\
&= (1-\frac{e-1}{n})(t+1) \\
&\quad + \frac{e}{n}(-nte^{-t} - ne^{-t} - e^{-t} + n + 1 - nte^{-1} - ne^{-1} + nte^{-t} + ne^{-t}) \\
&= (1-\frac{e-1}{n})(t+1) + \frac{e}{n}(-e^{-t} + 1 + n - nte^{-1} - ne^{-1}) \\
&= t - \frac{e-1}{n}t + 1 - \frac{e-1}{n} + (e-1) - t + \frac{e}{n} - \frac{e^{1-t}}{n} \\
&= e - \frac{t(e-1) + e^{1-t} - 1}{n}.
\end{aligned}$$

The best algorithm has this expectation minimized over $t \in [0,1]$. Verify that this is a concave function of t. Therefore the minimum is obtained at either $t = 0$ or $t = 1$. Since they have the same value, we have

$$\min_{t\in[0,1]} E[\sum C_j^{OL(t)}] = e - \frac{e-1}{n}.$$

Therefore, for any $t \in [0,1]$,

$$\frac{E[\sum C_j^{OL(t)}]}{E[\sum C_j^{OPT}]} \geq \frac{e - \frac{e-1}{n}}{e-1} = \frac{e}{e-1} - \frac{1}{n}.$$

The ratio can be made arbitrarily close to $\frac{e}{e-1}$, by choosing n large enough. The observation that it is useless for any algorithm to start the first job after time 1 shows that the above ratio holds for the best possible deterministic algorithm on this random instance.

Some of you might object that we do not prove a lower bound on $E[\frac{\sum C_j^{OL(t)}}{\sum C_j^{OPT}}]$, but the following lemma shows that what we showed is in fact enough.

Lemma 0.6. *Given an on-line optimization problem, with possible input sequences $\mathcal{I}$, and possible algorithms $\mathcal{A}$, both possibly infinite, for any random sequence I_p, and any randomized algorithm A_q, we have*

$$\min_{A\in\mathcal{A}} \frac{E_{I_p}[Z^A(I_p)]}{E_{I_p}[Z^{OPT}(I_p)]} \leq \max_{I\in\mathcal{I}} \frac{E_{A_q}[Z^{A_q}(I)]}{Z^{OPT}(I)}$$

provided that the left hand side is bounded.

Proof. Suppose that there exists a randomized algorithm A_q with competitive ratio c. Then $\forall I \in \mathcal{I}$

$$cZ^{OPT}(I) \geq E_{A_q}[Z^{A_q}(I)].$$

Hence,

$$\begin{aligned} cE_{I_p}[Z^{OPT}(I_p)] &\geq E_{I_p}[E_{A_q}[Z^{A_q}(I_p)]] \\ &= E_{A_q}[E_{I_p}[Z^{A_q}(I_p)]] \\ &= \min_{A \in \mathcal{A}} E_{I_p}[Z^{A}(I_p)]. \end{aligned}$$

□

Exercises

Exercise 1. Consider the following scheduling problem with *set-up times*:

Given is a set of n jobs and a set of m machines. Each job has a processing time p_j. Jobs may be preempted and, even stronger, several machines can work on any job simultaneously. However, before each part of job j that is processed a fixed time s_j must be spend on preparing the machine for job j. You may assume that $s_j = s$ for all j. The objective is to minimize the total completion time.

For the online version of this problem, in which jobs have also (unknown) release dates, find a lower bound on the competitive ratio for deterministic online algorithms and design a deterministic algorithm and analyse its competitive ratio. If you like you may take two machines ($m = 2$). Do not worry if you do not find matching lower and upper bounds.

Material

During the second lecture I treated

- Game tree evaluation from [MR], Sections 2.1 and 2.2.3
- Online optimisation related to Section 13.1 and 13.3 from [MR]

- On-line Scheduling, based on the papers:

  C. Chekuri, R. Motwani, B. Natarajan, C. Stein, Approximation Techniques for Average Completion Time Scheduling, *SIAM Journal on Computing 31*, 2001, 146–166

  L. Stougie, A.P.A. Vestjens. Randomized algorithms for on-line scheduling problems: How low can't you go? *Operations Research Letters 30*, 2002, 89–96.

  Please first try to google these papers, but if you do not have access to these papers, feel free to send me an e-mail and I'll forward a copy to you.