# A PROBLEM IN SCHEDULING: YOUR TIME STARTS NOW...

*Andrew McGregor*
University of Pennsylvania
*andrewm@cis.upenn.edu*

*Abstract*

We consider a scenario in which we wish to compute a good schedule of jobs using the same machine upon which we will process the scheduled jobs. The jobs to be processed are sufficiently short that the time spent processing a job is comparable to the time spent scheduling that job. We consequently have an interesting trade-off between how much time we should spend *thinking* about making our job schedule good (with respect to a given measure) and how much time we should spend *doing* (or processing) our jobs.

## 0. *Introduction*

In a widely circulated puzzle (see [3] for the earliest reference known by the author), the four members of the rock group U2 are late for a concert whose venue lies on the other side of a rickety bridge. All four men begin on the same side of the bridge. You must help them across to the other side. It is night. There is one flashlight. A maximum of two people can cross at one time. Any party who crosses, either 1 or 2 people, must carry the flashlight with them. The flashlight must be walked back and forth; it cannot be thrown, etc. Each band member walks at a different speed. A pair must walk together at the rate of the slower man's pace: the singer takes one minute to cross, the guitarist takes two minutes, the bass player takes five minutes and the drummer takes ten minutes. (One can only assume that they are carrying their instruments.) What is the minimum time taken for a sequence of bridge crossings that takes all band members to the other side of the bridge?

Anecdotal evidence suggests that, when attempting this problem most people see a 19 minute solution almost instantaneously but then take more than 2 minutes finding the optimal solution of 17 minutes.[1] This raises the question that, had the problem

[1] In the unlikely event that the reader is unfamiliar with the schedule that realizes the 17 minute solution, it is left as a potential diversion from reading this paper.

solver been there on the fateful night in question, how useful would he have been had he simply focussed on finding the optimal solution?

This phenomena, that of spending too long *thinking* and not enough time *doing*, carries across to computer science in numerous ways[2]. In this paper we look at a scheduling problem (see [2, 4] for comprehensive surveys of the large field of scheduling theory) whose jobs are sufficiently short that we are forced to consider time spent finding schedules on the same scale as time spent processing jobs that we've scheduled. In this problem, the concern that we spend excessive time thinking and not doing, will be abundantly clear. Such a scheduling problem could quite plausibly arise in a micro-architectural context.

## 1. *The Problem*

Consider the following familar scheduling scenario: You have available a single machine for the next $T$ seconds. There is a set of jobs $J = \{j_1, \ldots j_n\}$ you are considering running. Each job $j_i$ has a release time $r_i$, length $l_i$ and payoff $w_i$. We receive the payoff for a job iff we start the job after $r_i$ and complete it before time $T$. We seek a subset of jobs that we can feasibly schedule to yield the greatest payoff. It can be proven that we may assume w.l.o.g. that our machine is nonpreemptive.

This is a well understood scheduling problem for which there exists at least two pseudo-polynomial algorithms (running time $O(n \sum l_i)$ or $O(n \sum w_i)$) as well as a PTAS (running time $O(n^4/\epsilon)$). (See [1] for details.) However, implicit in all these scheduling algorithms is the assumption that either the schedule is computed prior to the $T$ seconds during which we can schedule jobs or else the time spent computing the schedule is incomparably small compared to the lengths of the jobs being scheduled. We make no such assumption. Furthermore we will compute the schedule using the same machine upon which we will process the scheduled jobs.

We compute for $T_1$ seconds and then schedule jobs for the remaining $T_2 = T - T_1$ seconds. We will compare our solution's payoff to the payoff of the optimum schedule of jobs in $[0, T]$. Note that this may seem a little unfair as we are in effect allowing the optimum algorithm to pre-compute the schedule before the time interval whereas our algorithm has to compute its solution during the $[0, T]$. This can give the optimum an arbitrarily large advantage: suppose $J$ consists of one very profitable length $T$ job released at time 0 and a collection of other very low profit jobs. We wouldn't be able to schedule the highly profitable job since even the slightest fraction of a second spent identifying this large job would mean it no longer fitted in the remaining available

[2]Least of which is the perceived wisdom that the computer itself is simply a machine on which the user spends hours saving time.

time. To ameliorate the situation, as a conciliatory gesture to algorithms that don't get to pre-compute, we assume that the length of the longest job in the optimal solution, $l^*$, is small compared to $T$, i.e. $l^* \leq T/k$ for some $k > 2$.

## 2. *A Solution*

Assume $r_1 \leq r_2 \leq \ldots r_n$, i.e. the jobs are given to us in order of increasing release time. We choose to primarily calculate our schedule via a dynamic program similar to one given in [1]. Let

$$D[i,t] = \begin{array}{c}\text{Optimal feasible schedule for time interval} [T-t, T] \\ \text{using jobs} \{i, i+1, \ldots n\}\end{array}$$

and, for ease of notation, let $D(t) = D[1,t]$. The dynamic programming table can be computed using the following recurrence

$$D[i,t] = \max\{D[i+1,t], D[i+1, \min\{T-r_i, t\} - l_i] + w_i\} \tag{1}$$

where $D[\cdot, t] = -\infty$ for $t < 0$. Let $D_S[i,t]$ be the schedule that achieves payoff $D[i,t]$. The recurrence follows from the fact that the optimum schedule can be achieved with scheduled jobs being processed in the ascending order of release times. This can be verified using an interchange argument.

We wish to quantify in seconds the running time of this algorithm so, to this end, we assume that calculating $D[i,t]$ from $D[i+1,t]$ and $D[i+1, \min\{T-r_i, t\} - l_i]$ using equation (1) takes $t_1/n$ seconds for some constant $t_1$. (We consider $t_1$ as part of the input to the problem.) Hence to compute the optimal schedule for the last $T_2$ seconds, $D[1,T_2]$ can be computed in time $t_1 T_2$ using the recurrence in equation (1).

Before we go any further we establish the following bound on the progress (in terms of payoff) of the dynamic program.

**Lemma 1**

$$\frac{D(t)}{D(T)} \geq \frac{t}{T}\frac{t-l^*}{t}$$

**Proof:** Consider the optimum schedule $J^* = \{j_1^*, j_2^*, \ldots\}$ where each scheduled job $j_i^*$ is started at time $s_i$ and ends at time $e_i$. Recall that of the jobs scheduled in the optimum schedule, they are processed in order of increasing release times. We partition the time interval $[0,T]$ into the following intervals

$$L_0 = [l_0, T] \text{ where } l_0 = \min\{T-t, \min\{s_k : e_k > T-t\}\}$$

$$L_i = [l_i, l_{i-1}] \text{ where } l_i = \min\{T - l_{i-1} - t, \min\{s_k : e_k > T - l_{i-1} - t\}\}$$

Now since $|L_i| \geq t$ for all $i$ we have partitioned $[0, T]$ into at most $T/t$ intervals. We now argue that in each interval the total weight of jobs scheduled is at most $D(t)\frac{t}{t-l^*}$. Consider the set of jobs in $L_i$. We remove jobs from $L_i$ in order of increasing weight to length ratios until all jobs can feasibly be scheduled in the interval $[l_{i-1} - t, l_{i-1}]$. This has decreased the weight of the jobs scheduled in $L_i$ by at most a fraction $\frac{t-l^*}{t}$. But the resulting set of jobs can be scheduled in time $[T - t, T]$ and hence

$$\frac{t - l^*}{t} \text{ (Weight of jobs scheduled in } L_i) \leq D(t)$$

and so $D(T) \leq \frac{T}{t} D(t)\frac{t}{t-l^*}$ as required. □

Using the proposed dynamic program, the more time we spend computing, the greater the interval $[T - T_2, T]$ we have scheduled. One might be tempted to compute until we have optimally scheduled $[0, T]$ and find what the optimal schedule would be. But this would be entirely academic as we would no longer be able to implement this schedule as a consequence of all the time we've spent finding the schedule. Hence, when we've spent $T_1$ time computing, it doesn't make any sense to schedule for more than $[T_1, T]$. This is the first observation to ensure that we don't waste time computing.

Conversely why would we want to stop computing at time $T_1$ if at this time we've only scheduled $[T - T_2, T]$ and $T_1 + T_2 < T$? If we did, we'd have scheduled nothing between $T_1$ and $T - T_2$ and the machine would be idle, waiting for our first scheduled job to start. Hence, we may as well continue computing optimal schedules for the increasing interval at the end of the $T$ seconds. In summary, in lieu of any other ideas the scheduler should just systematically and optimally build up its dynamic program table up until the point when it has no time to do anything other than implement it's best schedule so far computed. We refer to this scheduling algorithm as the "Wisely and Slow" algorithm.

Let $T_1^*$ be the optimum length of time to spend computing. In light of the above discussion $T_1^* = \frac{T}{1+1/t_1}$ and we construct an optimal schedule for $[T_2^*, T]$ where $T_2^* = \frac{T}{t_1+1}$. Appealing to Lemma 1 gives us the following bound on the performance of this algorithm.

**Theorem 1** *Using the Wisely and Slow Algorithm we achieve the following approximation to the best schedule of* $[0, T]$.

$$\frac{1}{t_1 + 1}\left(1 - \frac{t_1 + 1}{k}\right)$$

## 3. *A Better Solution*

Consider the situation in which we have spent 20 seconds scheduling the final 40 seconds of our allotted 100 seconds for which we have our machine. (Here $t_1 = 0.5$.) The previous algorithm would continue scheduling another 26 seconds such that we end up getting the best payoff that is possible by scheduling jobs in the last 66 seconds. But what if there were a quick and dirty (not necessarily optimal) way to augment the existing schedules for the final 40 seconds to creat a schedule for the last 80 seconds? It may be the case that running jobs according to a quick and dirty schedule for 80 seconds is better than running an optimal schedule of jobs for 66 seconds. The benefit of a quick and dirty augmentation is the essence of the better solution we present in this section.

For this solution, instead of computing for $T_1$ seconds such that $T_1 + T_2 = T$ we stop computing when $T_1 + T_2 = T - l^* - t_1$ (recall that $l^*$ is the length of the longest job). At this point, for all $i \in [n]$ we compute

$$E[i, T_2] = \max\{E[i+1], D[i+1, A] + w_i\}$$

where $T - A = T - \min\{T_2, T - \max\{r_i, T - l^* - T_2\} - l_i\}$ is the earliest time at which we can schedule a job after $j_i$ given that $j_i$ is scheduled and starts after time $\max\{r_i, T - l^* - T_2\}$.

Note that the computation of $E(T_2) = E[1, T_2]$ requires the same amount of time as computing a column of $D[\cdot, \cdot]$, i.e. about $t_1$ seconds. Hence, by time $T - T_2 - l^*$ we have computed $E(T_2)$ and in the remaining $T_2 + l^*$ seconds we still have time to implement this schedule.

The following lemma gives us a bound on $E(T_2)$.

**Lemma 2**

$$\frac{E(T_2)}{D(T)} \geq \frac{T_2}{T}$$

**Proof:** Consider the optimum schedule $J^* = \{j_1^*, j_2^*, \dots\}$ where each scheduled job $j_i^*$ is started at time $s_i$ and ends at time $e_i$. Consider an interval of time $L = [e_i - T_2, e_i]$ for some $i$. Let the jobs processed by the optimum in this interval be $j_\alpha^*, j_\beta^*, j_\gamma^* \dots j_i^*$. Note that at most one of these jobs, $j_\alpha^*$, is not entirely processed in the interval. Consider the total weight of these jobs $W = w_\alpha + w_\beta + w_\gamma + \dots w_i$. In order to prove the lemma we argue that $E(T_2) \geq W$ and then $E(T_2) \geq D(T)T_2/T$ follows.

Note that $l_\beta + l_\gamma + \dots + l_i \leq T_2$ since if $j_\beta^*, j_\gamma^* \dots j_i^*$ can all be scheduled in their entirety within an interval of length $T_2$. Since $l_\alpha \leq l^*$ we also have $l_\alpha + l_\beta + l_\gamma +$

$\ldots + l_i \leq l^* + T_2$. Lastly $l_\alpha + l_\beta + l_\gamma + \ldots + l_i \leq T - r_\alpha$ since if $j^*_\alpha, j^*_\beta, j^*_\gamma \ldots, j^*_i$ are all simultaneously schedulable, they are schedulable in the interval $[r_\alpha, T]$. Hence

$$l_\beta + l_\gamma + \ldots + l_i \leq \min\{T_2, T - \max\{r_\alpha, T - l^* - T_2\} - l_\alpha\}$$

Therefore

$$\begin{aligned} E(T_2) &\geq E[\alpha, T_2] \\ &= \max \left\{ \begin{array}{l} E[\alpha+1], \\ D[\alpha+1, \min\{T_2, T - \max\{r_\alpha, T - l^* - T_2\} - l_\alpha\}] + w_\alpha \end{array} \right\} \\ &\geq D[\alpha+1, \min\{T_2, T - \max\{r_\alpha, T - l^* - T_2\} - l_\alpha\}] + w_\alpha \\ &\geq w_\alpha + w_\beta + w_\gamma + \ldots + w_i = W \end{aligned}$$

□

This leads us to an algorithm where in the first $\frac{T - l^* - t_1}{1 + 1/t_1}$ seconds we patiently build up the dynamic programming table schedules for $[T - \frac{T - l^* - t_1}{1 + t_1}, T]$. Then in the next $t_1$ seconds we make a greedy choice of one more job to schedule. We call this the "Impetuous" scheduling algorithm (although to its credit, it is patient and methodical up until the last $t_1$ seconds.) Applying the result from the above lemma gives the following theorem.

**Theorem 2** *Using the Impetuous Algorithm we achieve the following approximation to the best schedule of* $[0, T]$.

$$\frac{1}{t_1 + 1}\left(1 - \frac{1}{k} - \frac{t_1}{T}\right)$$

## 4. *Conclusions and Further Work*

We have formulated a scheduling problem in which the phenomena of trading-off thinking with doing is made manifest to the algorithm designer. Various common sense principles are brought to bear upon the problem including those of "not wasting your time with things of only academic interest", "not sitting idle when there are jobs to be done" and "being greedy when there's no time to regret it isn't all that bad." In light of these principles two algorithms are presented that approximate the optimum schedule in our problem.

An interesting variant of the problem discussed would allow processing on two machines. An important concern in this scenario would be the waste of letting one

machine idle while the other machine computes schedules. (We would assume that parallel computation of the schedules is unfeasible.) The question then is how to put the first machine to work while we are scheduling on the second. The difficulty lies in the fact that identifying optimum jobs to process on the first machine in this interval would likely require knowledge of the schedule we are in the process of computing on the second machine.

Consider the following example: We wish to schedule the cooking of a large meal with the goal of cooking the "best" selection of component dishes subject to the constraint that we finish the preparation within an hour. We may now have precedence constraints in addition to release times of the jobs, e.g. the time by which the chicken will have thawed. As above, we have the trade-off between the optimality of the schedule we think up and the length of time we spend thinking it up. In the interest of not wasting as much time while thinking, we may *multi-task* and boil some water (the cooker is machine 1) while thinking about what to cook (using machine 2, the cook's brain) even though we have not necessarily yet determined we need boiled water and aren't better utilizing the cooker by putting on the sprouts. What are the principles that governed the choice of boiling water over sprout cooking?

Finally, the reader is cautioned about spending too long trying to solve these problems. A particularly wise wizard of Middle Earth once said "All you have to do is decide what to do with the time given to you." The reader has to decide whether the above problems merit their thought in the time given to them. However, in light of the above discussion, it is best not to spend *too* long making this decision.

## *Acknowledgments*

Thanks to Sampath Kannan for some helpful discussions and numerous friends who would calculate $D[1, T]$, for inspiration.

## *References*

[1] D. Karger, C. Stein, and J. Wein *Scheduling Algorithms.* Chapter in CRC Handbook on Algorithms. CRC Press, 1998.

[2] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *Sequencing and scheduling: algorithms and complexity.* In S.C. Graves, A.H.G. Rinnooy Kan, and P.H. Zipkin, editors, Handbooks in Operations Research and Management Science, volume 4. Elsevier, Amsterdam, 1993

[3] S. Levmore, and E. Cook. Super Strategies For Puzzles and Games. Doubleday & Company, Inc, Garden City, New York 1981

[4] M. Pinedo. *Scheduling: Theory, Algorithms and Systems.* Prentice Hall, 1997.