Scheduling

Subhash Suri

February 6, 2020

1 Time Management

- Scheduling is about *time management*: how to allocate a precious resource (e.g. time) to work on *which* tasks and *when*?
- Origin story: In 1874, Frederick Taylor turned down admission to Harvard to become an apprentice machinist at Enterprise Hydraulic Works in Philadelphia. Afterwards, he joined Midvale Steel Works, rising through his ranks until becoming its chief engineer.
- At Midvale, Taylor realized that time of the machines (and people) was not being used very well, and created a "planning office," at the heart of which was a bulletin board displaying the shop's schedule for all to see. *The board showed every machine in the shop, along with the task currently being processed, and all the tasks waiting for it.*
- Later, Henry Gantt will go on to build on this practice to develop the famous Gantt charts, which would help organize some of the most ambitious construction projects, including *Hoover dam*, *interstate highway system*, and are still used by firms like *Amazon, Ikea, and SpaceX.*
- Taylor and Gantt made scheduling an object of study, but their primary focus was on visualizing and conceptualizing all the information, not the fundamental problem of determining *which schedules were the best.*
- The mathematical aspects of scheduling have their origins in RAND Corporation's mathematician Selmar Johnson's 1954 paper, where he considered optimal bookbinding: each book needs to be printed on one machine and then bound on another.
- A more common instance of this 2-machine problem occurs at home: the laundry.
    1. Each load of laundry requires two steps: washing and drying, each performed on a separate machine.

2. Each load, depending on its content, may require very different time to complete, and so either the washer or the dryer may often be idle.
3. So, here is a scheduling problem: given n laundry loads, where the ith load requires A_i time on the washing machine, and B_i time on the dryer, in which order should they be done to minimize the total time needed to complete all the loads?

- Johnson proposed the following algorithm: of all the remaining tasks, find the shortest one (washing or drying). If this task it washing, do that load *first*; if this task is drying, do that load *last.* Repeat this process until all the loads are finished.
- **Homework Exercise.** Prove the correctness of this algorithm.
- Johnson's work, beyond just its immediate applications, made two deeper contributions:

  1. first, the scheduling could be expressed as an *algorithmic* problem, and
  2. second, it was possible to find *optimal schedules.*

  This work kicked off an enormous, and still growing, research literature for all conceivable types of scheduling problems.
- Let us explore the scope, subtlety, and complexity of scheduling algorithms by considering its simplest form: scheduling on a *single* machine.

2 Scheduling with Deadlines

- What is there to decide in a single-machine scheduling? If all the jobs need to be performed on one machine, the total amount of time needed is *fixed* irrespective of the schedule.
- Thus, the first important lesson in scheduling (in fact, in most algorithmic problems) is to *make the goal explicit.* We can't declare one schedule the best unless we have a clear metric for scoring different schedules. Indeed, *which metric we pick has a huge impact on the choice of the schedule, as well as the complexity of optimizing it.*
- A number of different metrics have been explored in scheduling, and let us begin with one of the early successes: *minimizing lateness.*

Minimizing Lateness

- In many applications, we can associate a *due date* with each job, and its lateness in a schedule is the amount of time past its due date (lateness) when it is completed.
- For example, if you have jobs from many clients, each preferring its job to be done by certain deadline. You would like to schedule the jobs in such a way that the *worst delay* (wait time) for any client is as small as possible.
- The optimal strategy for *minimizing lateness* is to start with the job with the *earliest deadline*, and work your way towards the task due last. This is called *Earliest Due Date* algorithm.
- Mathematically, suppose we have n jobs, where the ith job requires p_i processing time on the machine, and has a due date of d_i. The EDD algorithm will process these jobs in the increasing order of d_i's, *irrespective* of their processing times.
- **Homework Exercise.** Prove that EDD minimizes the maximum lateness, namely, $\max_i(\max\{0, c_i - d_i\}$, where c_i is the completion time for job i under EDD.
- **Assumption.** We assume that all the due dates and processing times are relative to a common clock, which starts at 0.

Maximizing Number of Completed Jobs

- A slightly different goal in scheduling might be to finish *as many jobs* as possible *before* their deadline. That is, we get credit for each job that is completed before its due date, and none for the job that is finished after its due date.
- **Homework Exercise.** Show that a slightly modified version of EDD is optimal for this version of scheduling.

Minimizing Total Delay

- Sometimes due dates are not a concern, and we just want to complete the tasks as quickly as possible. What does this mean? (In scheduling, as in many computer science algorithms, translating a simple desire into an explicit and precise metric is harder than it sounds.)
- In single-machine scheduling, the total time needed to complete all the tasks is the same no matter what schedule we use, so what do we mean by "finishing as quickly as possible"?

- In stead of the perspective of the "machine shop" if we adopt the perspective of the "clients" whose jobs are being processed, then each client has a different "completion time," namely, when it's job gets done.

- As an example, if we have two jobs, requiring processing times 4 and 1, then performing them in the order $(4, 1)$ leads to the wait time of 4 for the first client, and 5 for the second client, for a total of $4 + 5 = 9$ total wait time. Performing in the reverse order $(1, 4)$ leads to total weight time $1+5 = 6$. In this client-centric way, the second schedule is better.

- Suppose we have n jobs, where the ith job requires p_i processing time. The wait time for client i is w_i, the time at which ith job is completed. What is the optimal scheduling algorithm for minimizing $\sum_i w_i$?

- **Homework Exercise.** Show that Shortest Processing Time algorithm, which always schedules the shortest job first, is optimal.

- **Weighted Sum of Completion Times.** The input is a set of n jobs, where the ith job requires p_i processing time, and has a *weight* $\alpha_i > 0$. The wait time for client i is w_i, the time at which ith job is completed. What is the optimal scheduling algorithm for minimizing $\sum_i \alpha_i w_i$?

- **Homework Exercise.** Show that a modified Shortest Processing Time algorithm, which always schedules jobs in the *decreasing* order of α_i/p_i (relative importance), is optimal.

- Similar strategies appear in finance as well, for instance, in debt reduction. If you have many debts, then the debt avalanche strategy says to ignore the number of size of your debts, and simply funnel money towards the debt with the highest interest rate. If on the other hand you wish to reduce the *number* of debts, rather than the amount of debts, then you are back to the unweighted scheduling case—pay off the smallest debt first.

3 Dilemma of Choosing Metrics and Priorities

- With so many different ways to score a schedule, which one should one use? "A man with one watch knows what time it is; a man with two watches is never sure."

- Computer scientists are great at designing optimal algorithms for various metrics, but choosing which metric we should follow is up to us. In many ways, we get to decide *which problem we want so solve.*

- Indeed, the choosing of an appropriate metric is a *modeling* question, and asks us to decide what our priorities are: when there are many tasks to do, how should they be prioritized?

- Thinking about scheduling is also a good way to understand the phenomenon of "procrastination," the classic pathology of time management. Let us consider two illustrative stories about scheduling:

- **Pre-Crastination:** In an experiment at Penn State: participants were asked to bring either of the two heavy buckets to the opposite end of a hallway. One of the buckets was right next to the participants; the other was partway down the hall.

- To the experimenter's surprise, people immediately picked up the bucket next to them, and lugged it all the way down—passing the other bucket on the way—which they could have carried only a fraction of the way.

- Researchers called this seemingly irrational tendency the "pre-crastination:" hastening of subgoal completion.

- Focussing *not* just on getting things done, but on getting *important* things done should prevent procrastination. But, as the following story shows, handling priorities (weights) in scheduling correctly can be tricky.

- **Priority Inversion:** In 1997, Mars Rover, landed on the rocky red Martian surface, to explore the surface and collect data, after a 300 million miles journey.

- It had an on-board scheduler. However, instead of executing its highest priority task, of moving data into and out of its "information bus," the rover was mysteriously wasting its time on tasks of middling importance. JPL was stumped.

- When the Pathfinder's system noticed that the information bus had been idle for *unacceptably long time*, it initiated a complete restart, costing the mission many precious hours. A day later, the same thing happened again.

- Working feverishly on earth, the JPL time finally managed to reproduce and diagnose the problem. The culprit was something called *priority inversion.*

- Priority inversion occurs when

  - a low priority task takes possession of a system resource (in this case, database access) to do some work; but is then interrupted midway by a timer, which pauses it and invokes the scheduler;

  - the scheduler tees up the high priority task, but it can't run because it needs the resource (database), which is tied up by the paused job.
  - the scheduler then moves down the task list, running various unblocked medium-priority jobs, never getting the high-priority job to run, which is blocked by the low-priority job, or the low-priority job that is blocking the high-priority job but is itself stuck behind medium priority jobs.

- The fix was easy: *priority inheritance*, when a low-priority job is blocking a high-priority job then that low-priority job momentarily becomes the highest priority job.

- When a task cannot be started until another one is finished, scheduling theorists call that a *precedence constraint.* Naturally, in many scheduling applications, precedence constraints are an important factor, so let us now consider how to incorporate them into our setup.

4 Scheduling With Precedence Constraints

- The Earliest Due Date algorithm minimizes the *maximum lateness* among the jobs. What happens if the jobs also have *precedence* constraints among them?

- With precedence constraints, one cannot simply schedule the next job with the earliest due date if some jobs have to wait for other to finish before being started.

- In 1968, E. Lawler proved that this can be fixed by constructing the schedule in *back-to-front* order:

  > look only at tasks that no other task depends on, and put the one with the *latest due date* at the *end of schedule*. Then simply repeat the process, again considering at each step only those tasks that no other (as-yet unscheduled) task depends on.

- **Homework Exercise.** This seems intuitively correct. Prove that this back-to-front EDD minimizes the max lateness for scheduling with precedence constraints.

- But as Lawler looked more deeply into precedence constraints, he ran into difficulties. For the *Shortest Processing Time* algorithm, which minimizes the *sum of completion times* of jobs, or equivalently the *average* completion time, there does not appear to be a simple or obvious tweak to accommodate precedence constraints.

- Although it appeared to be an elementary scheduling problem, neither Lawler nor any of the many other esteemed CS researchers could find an efficient way to solve it.

- The difficulty turned out to be much deeper: Lawler ended up proving that under precedence constraints, the problem is *intractable*, namely, NP-hard!

- To put this in context, while in problems such as optimal stopping, some versions such as triple-or-nothing do not have a *solution*, *every scheduling* problem obviously does have an optimal solution—some schedule that is the best. The difficulty here is that there is no simple, namely, *polynomial time* algorithm to find that schedule.

- In scheduling, it has turned out that even the subtlest changes to a problem can often tip the scale and turn a nice simple problem into an intractable one. For instance, Moore's algorithm minimizes the number of *late* tasks when they have equal value (importance). But if jobs have unequal values (weights), then minimizing the weighted lateness is NP-hard.

5 Preemption

- In our study of scheduling so far, we have implicitly assumed a *non-preemption* policy—one a job is started on a machine, it runs to completion. But what happens if we allow pre-emption, letting the scheduler to pause (preempt) a running job, switch to another one, returning to the preempted job sometime later in the future.

- In many cases, preemption simplifies the scheduling problem considerably. In fact, the problems of minimizing *maximum lateness* or minimizing *sum of completion times* are both intractable with precedence constraints, but they return to being tractable with preemption.

- In fact, EDD and SPT, resp., remain the best algorithms for them with fairly straightforward modifications.

    1. When a task's starting time comes up, compare that task with the one currently running.
    2. For EDD, if the new task is due even sooner than the current one, switch; otherwise stay.
    3. For SPT, if the new task can be finished faster than the current one switch; otherwise stay.

- **Homework Exercise.** Prove correctness of these modified EDD and SPT algorithms.

- In fact, EDD and SPT are *optimal* algorithms even if we do not know all the jobs in advance. When new jobs arrive, under preemption, these algorithms still guarantee the best schedules.

- In fact, the *weighted* version of Shortest Processing Time is a pretty good candidate for best *general purpose* scheduling strategy in the face of uncertainty.
- It offers a simple prescription for optimal time management: when a new job arrives, divide its importance (weight) by the time it takes to complete. If that figure is higher than the job currently being processed, switch to the new one.
- Thus, preemption can be a powerful friend of scheduling. However, *preemption isn't free.* Every time we switch jobs, there is a price, often called *context switch* cost.

6 Multiprocessor Scheduling: Algorithms & Analysis

- We now analyze the theoretical performance of some NP-hard scheduling problems. We will consider only *non-preemptive* scheduling, where a job once started is performed until completion, without interruption.
- A particularly simple algorithm and analysis belongs to a problem called *minimum makespan* scheduling or *multiprocessor* scheduling. Informally, we have a set of m identical machines and our goal is to schedule a set of n jobs to minimize the finishing time of the last jobs.

6.1 Minimizing Makespan

- Formally, given a set of m (identical) machines $M_1, \ldots, M_m$, and a set of n jobs, where job j needs p_j time for processing, the goal is to schedule the jobs on these machines so all the jobs are finished as soon as possible.
- That is, find the minimum time (called MakeSpan) T by which all jobs can be executed collectively.
- Let A_i be the set of jobs assigned to machine M_i. Then, M_i needs time

$$T_i \quad = \quad \sum_{j \in A_i} p_j$$

this is called the load on machine M_i. We wish to minimize

$$T \quad = \quad max_i\{T_i\},$$

which is called the makespan of the schedule.

- The decision problem is NP-complete: it's in NP because we can decide whether the makespan is T or not. For completeness, we can reduce subset sum to it (scheduling on two machines).
- We show that the following simple greedy method gives good approximation. The algorithm makes one pass over the jobs in any order, and *assigns the next job j to the machine with the lightest current load.*
- for $j = 1, \ldots, n$
  1. Let M_i be the machine with the minimum T_i among the machines
  2. Assign j to M_i
  3. $A_i \;=\; A_i \cup \{j\}$
  4. $T_i \;=\; T_i + p_j$
- This algorithm is called a *list scheduling* algorithm.
- For instance, given 6 jobs, with sizes $2, 3, 4, 6, 2, 3$, and 3 machines, the algorithm gives

  $$(2, 6), (3, 2), (4, 3),$$

  for the makespan of 8. The optimal makespan is 7: $(3, 4), (6), (2, 2, 3)$.

6.2 Analysis of List Scheduling Algorithm

- Let T be the makespan produced by the algorithm, and T^* the optimal.
- Note that, unfortunately, we do not know the value of T^*. Nevertheless, we can use a lower bound for T^* to achieve an approximation guarantee.
- A simple lower bound comes from the following trivial observation: since there is a total of $\sum_j p_j$ amount of work to be done by the m machines collectively, the makespan cannot be less than the avg load. Thus,

  $$T^* \;\geq\; \frac{1}{m} \sum_j p_j \qquad (1)$$

- Unfortunately, this lower bound itself can be too weak: for instance, if we have one extremely long job, then the best we can do is to put it by itself on a single machine, but still the makespan has to be as long as this job. This greedy solution is optimal,

but the lower bound may be way off, if the remaining jobs are all very small. But this suggests another lower bound:

$$T^* \geq max_j\{p_j\} \qquad (2)$$

- **Theorem:** The greedy makespan produces an assignment with makespan $T \leq 2T^*$.
- **Proof.** Look at the machine that attains the max load (makespan). Let this be M_i.
  1. Let j be the last job assigned to M_i.
  2. Key Observation: When j was assigned, M_i was the machine with the least load! Its load just before assignment of j was $T_i - p_j$.
  3. Since this was the lightest load, *every* other machine has load at least $T_i - p_j$. Thus, adding up all these load, we have

$$\sum_{k=1}^{n} p_k \quad \geq \quad m \times (T_i - p_j)$$

     equivalently,

$$T_i - p_j \quad \leq \quad \frac{1}{m} \sum_{k=1}^{n} p_k$$

  4. But the sum on the right is just the sum of all the jobs, since each job is assigned to some machine. The quantity on the RHS is just the lower bound from (1), and thus

$$T_i - p_j \leq T^*$$

  5. Now we account for the last job. Here we simply use the inequality (2), which says that $p_j <= T^*$. Thus, we have the claimed approximation bound:

$$T_i = (T_i - p_j) + p_j \quad \leq 2T^*$$

- The following example shows that this bound is (almost) tight for the list scheduling algorithm.
  1. We have $m(m-1)$ jobs with processing time 1 and one job with processing time m.
  2. Suppose the greedy algorithm schedules all the short jobs before the long job.
  3. Then the makespan of the greedy schedule is $(2m-1)$ while the optimal makespan is m.
  4. Thus, the algorithm gives a schedule whose makespan is $(2 - 1/m)$ times the optimal.

6.3 An Improved Algorithm

- In the previous example, a better schedule is possible *had we processed the jobs in (decreasing) sorted order.* A natural modification of list processing, therefore, is the following algorithm.

    1. Sort the jobs in the *decreasing* order of processing time.
    2. For $i = 1$ to n do
    3. Assign job J_i to the least-loaded machine and update its load.

- The analysis is based on the following observation: if we have fewer than m jobs, then the algorithm is clearly optimal, since it puts each job on its own machine. If there are more than m jobs, then we can use the following lower bound on the optimum.

- **Lemma.** If there are more than m jobs, then $T^* \geq 2p_{m+1}$.

    1. **Proof.** Consider the first $m + 1$ jobs; they are in sorted order and each takes at least p_{m+1} time. The optimal must assign at least two of them to the same machine, and thus must have $T^* \geq 2p_{m+1}$.

- **Theorem.** Modified List Scheduling algorithm achieves makespan $T \leq \frac{3}{2}T^*$.

    1. **Proof.** Suppose the machine with the maximum load is M_i. If M_i has only one job, then the schedule is optimal. So, let's assume that M_i has at least two jobs.
    2. Let $j \geq m + 1$ be the last job assigned to M_i. Thus, $p_j \leq p_{m+1} \leq \frac{1}{2}T^*$.
    3. We now proceed as in the previous proof, with the following minor change. In the old proof we had $T_i - p_j \leq T^*$ and $p_j \leq T^*$, and we added them up to get the factor 2. In our case, the second inequality is $p_j \leq \frac{1}{2}T^*$, and so when we add them up we get

$$T_i = (T_i - p_j) + p_j \leq \frac{3}{2}T^*$$

- Multi-processor scheduling is NP-hard by reduction from 2-PARTITION. That, however, only shows that the problem is weakly NP-hard, which is indeed the case when the number of machines is a constant.

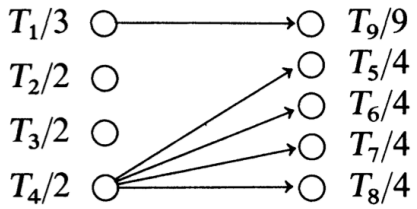
- When the number of machines m is arbitrary, the problem is also strongly NP-hard by reduction from 3-PARTITION.

7 Anomalies in Scheduling

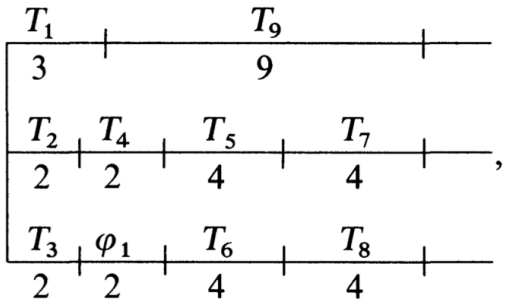
- The list scheduling algorithm for multiprocessor is both amazingly simple and fairly efficient, considering its intractable complexity.
- This simplicity, however, hides some surprising and counter-intuitive behavior. These "anomalies" have been observed by researchers in multi-processor systems or parallel computing, and it's worth studying them.
- Our discussion follows the exposition in the following paper of Ron Graham: *Bounds on multiprocessor timing anomalies*, SIAM Journal of Applied Math 1969. `https://pdfs.semanticscholar.org/f389/c27fec10f5df988c96e4d145c9dde955b876.pdf`.
- We have n identical processors called $P_1, P_2, \ldots, P_n$ and r tasks $T_1, T_2, \ldots, T_r$.
- A processing time function $\mu : T \to (0, \infty)$ prescribes how much time each task needs for completion on a machine.
- A partial order $\prec$ specifies precedence constraints between jobs, which demands that if $T_i \prec T_j$, then T_j cannot begin until T_i has been completed.
- In order to explore anomalous behavior of list scheduling, we assume a list L, called *priority list*, containing all the tasks, is given. Every time a processor becomes free, it selects the first open (unscheduled) task T_j from L that is *ready*, and then executes it for $\mu(T_j)$ units of time. (A ready job is one that has no unfulfilled precedence constraint.)
- In the following illustrations, we indicate the partial order as a directed graph, where (T_i, T_j) is an edge whenever $T_i \prec T_j$.
- For each task T_j, we write $T_j/\mu(T_j)$ to show its processing time.
- The schedule for each processor is shown as a timing diagram, along the time axis.
- The first figure shows the schedule for $n = 3$ machines and 9 jobs, with a makespan of $\omega = 12$.
- The completion time is clearly a function of four parameters: $L, \mu, \prec, n$. In the following, we vary each of these parameters and see how this affects ω.
- We have already seen that changing the ordering of jobs in the list L affects the makespan, which is not surprising. Figure (i) shows a rearrangement L', which increases the makespan to 14.

Example. $n = 3$; $L = (T_1, T_2, \cdots, T_9)$.

$G(\prec, \mu)$:



D:



, $\omega = 12.$

- However, suppose we keep the list L fixed, but *tip the scale* in favor of the scheduler by *increasing* the number of machines, *decreasing* the number of constraints, or *decreasing* the processing times of the jobs. We should expect the schedule *using the same fixed* L to only improve, right?

7.1 The Anomalies

- The following examples show how the makespan changes, as we *relax* the scheduling constraints, while keeping the sequence L fixed.
- **Changing $\prec$.** We drop two of the precedence constraints.
- **Decreasing μ.** We reduce processing times as $\mu'(T_i) = \mu(T_i) - 1$, for all i.
- **Increase number of machines.** Lastly, we increase n from 3 to 4.
- The following figures illustrate the changes in ω in response to these modifications.
- Contrary to what might be generally expected, *relaxing* $\prec$, decreasing μ, and increasing n can all cause ω to *increase*.

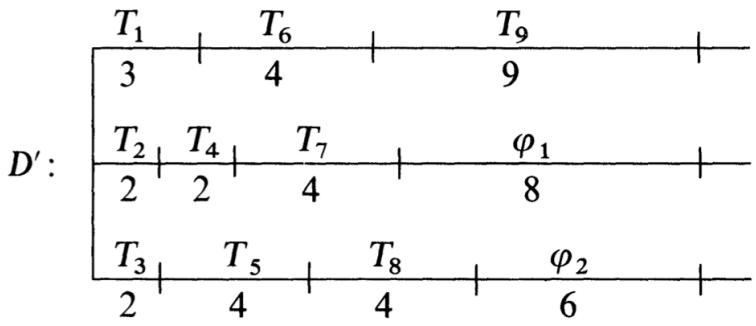
(i) Replace L by $L' = (T_1, T_2, T_4, T_5, T_6, T_3, T_9, T_7, T_8)$, leaving μ, $\prec$ and n unchanged. In this case we obtain

D':

T_1	T_3	T_9	
3	2	9	
T_2	T_5	T_7	φ_1
2	4	4	4
T_4	T_6	T_8	φ_2
2	4	4	4

and $\omega' = \omega'(L', \mu, \prec, n) = 14$.

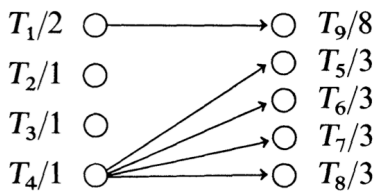
(ii) Change $\prec$ to $\prec'$ by removing $T_4 \to T_5$ and $T_4 \to T_6$. Then



and $\omega' = \omega'(L, \mu, \prec', n) = 16$.

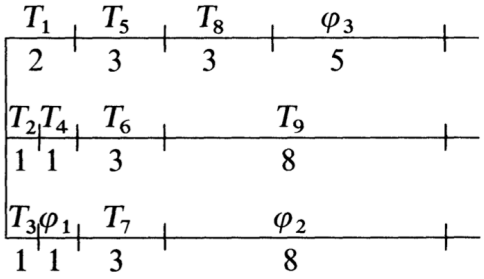
and $\omega = \omega(L, \mu, \prec, n) = 16$.

(iii) Decrease μ to μ' by defining $\mu'(T_i) = \mu(T_i) - 1$ for all i. In this case $G(\prec, \mu)$ becomes



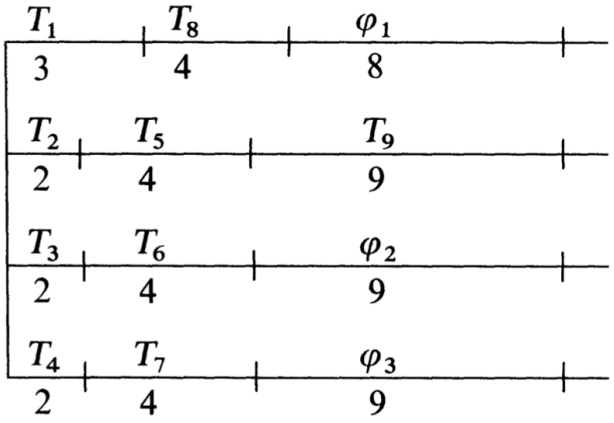
and

D':



with $\omega' = \omega'(L, \mu', \prec, n) = 13$.

(iv) Increase n from 3 to 4. Then



and $\omega' = 15$.