Scheduling
Lecture 1: Scheduling on One Machine

Loris Marchal

October 16, 2012

1 Generalities

1.1 Definition of scheduling

- allocation of limited resources to activities over time
- *activities*: tasks in computer environment, steps of a construction project, operations in a production process, lectures at the University, etc.
  tasks, jobs,
- *resources*: processors, workers, machines, lecturers, rooms, etc.
  processors, machines
- *objective*: minimize total time, energy consumption, average service time

Many variations on the model, on the resource/activity interaction and on the objective.

- Typical example: organize production of a workshop, by making a schedule for machines. Making different objects may need different series of machines, with different processing time. The number of objects of each type (or the ratio) may be fixed, and we want to minimize the overall processing time (or the throughput).
- Another example: in an airport, allocate gates to airplanes, arriving at different gates, not all gates can accomodate all planes, some gates are further than others: how to minimize the time spent by passengers (taxiing and walking in the terminal).
- Other examples: schedule the activity of workers in a construction process, allocate rooms for lecture at the University, etc.
- Our context: in a computing system, we have a number of machines at hand. These machines may be close to each other, or linked with a limited communication network. We want to use this platform to execute an application, composed of several tasks. Tasks may have precedence constraints, or other type of constraints. Several users may share the platform, which might be widely distributed, or hierarchical.

1.2 Graham notation

Classes of scheduling problems can be specified in terms of the three-field classification $\alpha|\beta|\gamma$ where

- α specifies the machine environment,
- β specifies the job characteristics,
- γ and describes the objective function(s).

We will illustrate this notation on all following scheduling problems.

2 Scheduling with a single machine

Motivations:

- understand the complexity of the problem for various objectives
- some of actual systems offers flexibility and may well be modeled by a single machine (e.g. virtual machines)
- practice usual scheduling techniques

Objectives using C_i:

- Makespan $C_{\max} = \max C_i$ **most common objective**
- Total *flow* time: $\sum_{j=1}^{n} C_j$
- Weighted (total) flow time: $\sum_{j=1}^{n} w_j C_j$

2.1 First example, with new objective, $1||\sum w_i C_i$, polynomial (Smith-ratio)

Let us consider the problem: $1||\sum w_i C_i$,

- 1 machine
- no constraints on tasks (length p_i)
- Objective: weighted sum of completion times

Exemple:

tasks	A	B	C	D
w_i	2	1	3	1
p_i	2	2	4	4
w_i/p_i	1	0.5	0.75	0.25

- Intuitions:
  - put high weight first (C, A, B, D, cost: $12 + 12 + 8 + 12 = 44$)
  - put longer tasks last (B, A, C, D, cost: $2 + 8 + 24 + 12 = 46$)
- $\Rightarrow$ Order task by non-increasing Smith ratio: $w_1/p_1 \geq w_2/p_2 \geq \cdots \geq w_n/p_n$ (on the example A,C,B,D, cost:$4 + 818 + 8 + 12 = 42$)

Proof:

- Consider a different optimal schedule S
- Let i and j be two consecutive tasks in this schedule such that $w_i/p_i < w_j/p_j$
- contribution of these tasks in S:
  $S_i = (w_i + w_j)(t + p_i) + w_j p_j$
- contribution of these tasks if switched:
  $S_j = (w_i + w_j)(t + p_j) + w_i p_i$

- we have
  $\frac{S_i - S_j}{w_i w_j} = \frac{p_i}{w_i} - \frac{p_j}{w_j}$
  Thus we decrease the objective by switching these tasks.

2.2 Adding due-dates

Other objectives in the Graham notations, using *due dates* d_j ((sometimes) appears in the job characteristics):

- *lateness*: $L_j = C_j - d_j$
- *tardiness*: $T_j = \max\{0, C_j - d_j\}$
- *unit penalty*: $U_j = 0$ if $C_j \leq d_j$, 1 otherwise

wich gives the following objectives:

- (total lateness=sum flow)
- maximum lateness: $L_{\max} = \max L_j$
- total tardiness $\sum T_j$
- total weighted tardiness $\sum w_j T_j$
- number of late activities $\sum U_j$
- weighted number of late activities $\sum w_j U_j$

2.2.1 EDF for $1||L_{\max}$

Let's study the problem $1||L_{\max}$: minimize the maximum lateness on one machine. The strategy which places first the jobs with the earliest deadlines is optimal (Earliest Deadline First).

Theorem 1. *EDF is optimal for* $1||L_{\max}$.

Proof. We assume that there exists an optimal schedule S which does not follow the EDF rule. We consider the first pair of consecutive tasks k, l processed in this order and such that $d_k > d_l$. We call S' the schedule obtained by exchanging k and l in S, and t the sarting time of k in S. We call L_i^S the lateness of task i in schedule S:

$$\begin{aligned} L_k^S &= t + p_k - d_k \\ L_l^S &= t + p_k + p_l - d_l = L_k^S + d_k - d_l > L_k^S \quad \text{since } d_k > d_l \end{aligned}$$

In schedule S':

$$\begin{aligned} L_l^{S'} &= t + p_l - d_l < L_l^S \\ L_k^{S'} &= t + p_k + p_l - d_k = L_l^S + d_l - d_k < L_l^S \end{aligned}$$

Thus, by exchanging k and l, we decrease their lateness and we do not change the other tasks. We can continue with the next pair of tasks which do not follow the EDF order, and after iterating this process, we end up with an EDF schedule with optimal $L_{\max}$. □

2.2.2 Moore-Hodgson algorithm for $1||\sum U_i$

Example:

job	1	2	3	4	5
d_j	6	7	8	9	11
p_j	4	3	2	5	6

Tasks are sorted by non-decreasing $d_i : d_1 \leq \cdots \leq d_n$

```
1. $A := \emptyset$
2. For $i = 1 \ldots n$
   (a) If $p(A) + p_i \leq d_i$, then $A := A \cup \{i\}$
   (b) Otherwise,
       i. Let $j$ be the longest task in $A \cup \{i\}$
       ii. $A := A \cup \{i\} - \{j\}$
```

Optimal solution : $A = \{2, 3, 5\}$

Proof.

- Feasibility:

  We first prove that the algorithm produces a feasible schedule, that is, all task of A can be scheduled in this order without missing their deadline

  - By induction: if no task is rejected, ok
  - Assume that A is feasible, prove that $A \cup \{i\} - \{j\}$ is feasible too
    - all tasks in A before j: no change
    - all tasks in A after j: shorter completion
    - task i: let k be the last task in A: $p(A) \leq d_k$
      since task j is the longest: $p_i \leq p_j$, thus $p(A \cup \{i\} - \{j\}) \leq p(A) \leq d_k \leq d_i$ (because tasks are sorted)
      That is, the new task i terminates earlier than k before j was rejected. Since $d_i \geq d_k$, this is enough.

- Optimality:

  Assume that there exists an optimal set O different from the set A_f output by the Moore-Hodgson algorithm

  - Let j be the first task rejected by the algorithm which is included in O (there exists such a task since O is different from A_f and the cardinality of O is larger or equal to the cardinality of A_f)
  - We consider the set A at the moment when task j is rejected from A, and i the task being added at this moment
  - $A \cup \{i\}$ is not feasible, thus O does not contain $A \cup \{i\}$
  - Let k be a task of $A \cup \{i\}$ which is not in O
  - Since the algorithm rejects the longest task, $p(O \cup \{k\} - \{j\}) \leq p(O)$
  - In the solution of O, we can replace j by k without increasing any completion time.

  We can repeat this process, until we get the set of tasks scheduled by the algorithm.

□

2.3 Adding release dates, $1|r_i|\sum C_i$

All jobs are not released at the same time, but job j is available starting at time r_j.

$1|r_i|\sum C_i$ is NP-complete (admitted)

When adding preemption, we can derive an optimal algorithm for $1|r_i, pmtb|\sum C_i$, which serves as a basis for a 2-approximation algorithm for the initial problem.

Theorem 2. *Shortest Remaining Processing Time first (SRPT) is optimal for* $1|r_i, pmtn|\sum C_i/$

Proof. Let us consider an optimal schedule S which does not follow the SRPT rule. This means that at a given time t a task k is being processed whereas there exists another task l with $x_l < x_k$ (x_i denotes the fraction of remaining work for task i at time t). Starting at time t, a time $x_k + x_l$ must be devoted to the pair of tasks k, l. We modify the schedule such that the first x_l time units of this time are devoted to l and the remaining x_k time units are devoted to x_k. Let us denote by S' the modified schedule. Since $x_l < x_k$, l finishes earlier in S' than k in S:

$$C'_l \leq C_k$$

Moreover, $C'_l < C_l$ since at time t, S processes task k and S' processes l. Thus, $C'_l = \min\{C_k, C_l\}$. Finally, $C'_k = \max\{C_k, C_l\}$ and nothing changes for the other tasks. Thus, the objective is decreased in S'. □

The algorithm A for the original problem (without preemption) is the following:

1. Solve the relaxed version with preemption using SRPT;
2. Sort the tasks by increasing completion time in this solution: $C_1^P < C_2^P < \cdots < C_n^P$
3. Process the tasks in this order without preemption. If a task is not yet available at given time, wait for it.

Definition 3. Consider a minimization problem. Let $OPT(x)$ denote the optimal value of the objective for any instance x. An algorithm A is a ρ-approximation algorithm if and only if for any instance x, $A(x) \leq \rho OPT(x))$.

Theorem 4. *The algorithm proposed above is a 2-approximation algorithm for* $1|r_i|\sum C_i$.

Proof. Instead of comparing directly to the optimal solution (which we do not know), we compare to a lower bound on this optimal solution:

$$\sum C_i \geq \sum C_i^P,$$

since the solution of the problem without preemption is a solution for the problem with preemption. We will now prove that for each task i, $C_i \leq 2C_i^P$.

Let us denote by t_j the time when j is started by A. We say the machine is *idle* at time t is it is not processing any task at time t. In our algorithm, it can only be idle if the next job to process has not yet be released. Let us denote by $idle(t)$ the total idle time of the machine before t. Note that

$$C_j = \sum_{i \leq j} p_i + idle(t_j).$$

First, $\sum_{i \leq j} p_i \leq C_j^P$ since the tasks are completed in the same order in the preemptive schedule.

Second, we now prove that there is no idel time from C_j^P to t_j (such that $idle(t_j) \leq C_j^P$). By contradiction, assume that the machine is idle at $t \in [C_j^P, t_j]$. This is possible only if the machine is waiting for some task k to be processed before j and which has not been released yet ($r_k > t \geq C_j^P$). However, the reason for this task to be processed before j is $C_k^P < C_j^P$. Thus $C_k^P < r_k$, which contradicts the assumption.

Thus $C_j \leq 2C_j^P$, and A is a 2-approximation algorithm. $\square$