CSEP 521 - Applied Algorithms

Scheduling Algorithms

Reading:

Scheduling Algorithms (1997)

David Karger, Cliff Stein, Joel Wein

(downloadable from course web-site)

Scheduling Theory

In general:

A set of jobs needs to be processed by a set of machines. The jobs need to be scheduled on the machines in a way that satisfies some objective function.

Scheduling Theory

Example 1a:

Given a set of jobs, each job has known processing-time and deadline. How do we schedule the jobs on a single machine in a way that minimizes the number of late jobs (those completed after their deadline).

Example 1b:

The same on two identical machines.

Example 1c:

The same on m machines each having a different processing rate.

Scheduling Theory

Example 2a:

Each exam needs to be marked by three teachers (each checking a different question). The order in which the questions are marked is not important.

For each exam and question, we know how much time it takes to mark it. What is the best schedule if we want to minimize the completion time of the whole marking process?

Example 2b: The same, but the questions must be marked in some fixed order.

Example 2c: The same, but now it takes some (known) time to transfer a set of exams from one teacher to another.

Scheduling Theory - Notations

A scheduling problem is defined by a triplet $\alpha|\beta|\gamma$.

Some possibilities for α:

1 - a single machine

P - identical parallel machines

Q - parallel machines with different rates.

R - unrelated parallel machines (specific processing time for each job and machines).

O – Open-shop scheduling

F – Flow-shop scheduling

Scheduling Theory - Notations

β - additional assumptions or constraints.

For example:

prmt- preemptions allowed

prec- precedence constraints

r_j – release times

γ- objective function. For example:

C_{max} – The makespan = last completion time.

$\sum_i C_j$ – sum of completion times (same objective as average).

More notations

J_j – The jth job.

p_j – length of J_j = how many processing units it requires.

r_j – release time of J_j = when does J_j is available for execution.

d_j – deadline (or due-date) for J_j = when do we need to complete its execution.

C_j – the completion time of J_j in a given schedule.

Algorithms for a Single Machine

We will see that simple greedy algorithms are optimal for some scheduling problems on a single machine.

Other problems, some of them look really simple, are NP-hard.

Shortest Processing Time (SPT) Rule

The problem: $1||\sum_j C_j$ (average completion time)

SPT Rule: Sort the jobs such that $p_1 \le p_2 \le \ldots \le p_n$.
Process the jobs according to this order.

Example: 5 jobs of lengths 9, 6, 3 ,8, 1

$\sum_j C_j$ in original order:

SPT order:

$\sum_j C_j$ in SPT order:

Shortest Processing Time (SPT) Rule

The problem: $1||\sum_j C_j$ (average completion time)

Theorem: SPT is optimal for $1||\sum_j C_j$
Proof:

$C_1 = p_1$; $C_2=p_1+p_2$; $C_j = \sum_{i \le j} p_i$

$\sum_j C_j = np_1 + (n-1)p_2 + (n-2)p_3 + \ldots + p_n = \boxed{\sum_j (n-j+1)p_j}$.

This is a product of two vectors. The first one (n,n-1,…,1) is decreasing. To get a minimal result, the other vector needs to be non-decreasing.

■

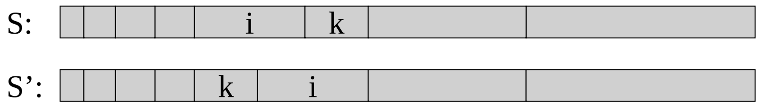
Optimality of SPT for $1||\sum_i C_j$

An alternative proof (using exchanging argument):

Assume that S is an optimal schedule which is not according to SPT.

For some pair J_i, J_k of adjacent jobs, J_k is scheduled after J_i while $p_i > p_k$.

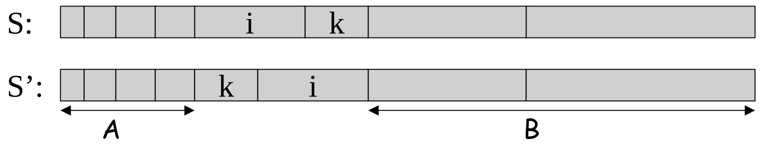
Build a new schedule S' in which we swap the schedules of J_i,J_k (the other jobs are as in S).



We show that S' is a better schedule:

Optimality of SPT for $1||\sum_i C_j$

Claim: In S' $\sum_j C_j(S') < \sum_j C_j(S)$.



Proof: A: jobs starting before J_i and J_k.
B: jobs starting after J_i and J_k.

$\sum_j C_j(S) = \sum_{j\in A} C_j + \sum_{j\in B} C_j + C_i + C_k =$
$= \sum_{j\in A} C_j + \sum_{j\in B} C_j + (p_A+p_i)+(p_A+p_i+p_k)$

$\sum_j C_j(S') = \sum_{j\in A} C'_j + \sum_{j\in B} C'_j + C'_i + C'_k =$
$= \sum_{j\in A} C_j + \sum_{j\in B} C'_j + (p_A+p_k)+(p_A+p_k+p_i) =$
$= \sum_j C_j(S) + p_k - p_i < \sum_j C_j(S)$ (since $p_i > p_k$).

■

Variants of SPT

1. The problem: $1|r_j, pmtn| \sum_j C_j$

Shortest Remaining Processing Time (SRPT) Rule:
At each moment, process the job with the shortest remaining processing time (can preempt a currently processed job).
Complexity: Should keep a sorted list of all available jobs. O(log n) for any released job + O(log n) for any preempted job. The total number of preemptions is at most n. → O(n log n) in total.
Theorem: SRPT rule is optimal for $1|r_j, pmtn| \sum_j C_j$
Proof: Exchanging argument.

Variants of SPT

2. The problem: $1|| \sum_j w_jC_j$

Weighted Shortest Processing Time (WSPT) Rule:
Sort the jobs such that $p_1/w_1 \leq p_2/w_2 \leq \ldots \leq p_n/w_n$.
Process the jobs on the machine according to this order.

Theorem: WSPT is optimal for $1|| \sum_j w_jC_j$

Proof: Exchanging argument.

Single Machine. Set-up times.

The problem: $1|\text{set-up}| C_{max.}$

For each pair of jobs, s_{ij} is the set-up time required between processing i and j.

Note: without set-up times, or with identical set-up times ($\forall i,j\ s_{ij} = s$), any order is optimal ($C_{max} = \sum_j p_j + (n-1)s$).

For arbitrary set-up times. The problem is as hard as the traveling salesman problem.
Can you see the reduction??

EDD for Minimizing Tardiness.

For an instance with due-dates and a given schedule
$L_j = C_j - d_j$ (Lateness)
$T_j = \max(0, L_j)$ (Tardiness)
Possible objectives: Minimizing T_{max}, L_{max}, $\sum_j T_j$, $\sum_j L_j$

EDD Rule (earliest due-date): Sort the jobs such that $d_1 \leq d_2 \leq \ldots \leq d_n$. Process the jobs on the machine according to this order.

Theorem: EDD is optimal for $1||T_{max}$ and $1||L_{max}$
Proof: exchanging argument.
Theorem: $1||\sum_j T_j$ is NP-hard.

EDD for Minimizing Tardiness.

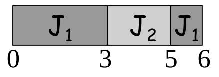
The problem: $1|r_j, pmtn|T_{max}$.

EDD rule: Process the job with minimal due-date among the jobs that are available.

- When a new job with an early due-date is released we might preempt the currently processed job.

Example: $r_1=0, p_1=4, d_1=6$.
$r_2=3, p_1=2, d_2=5$.

The EDD schedule:



No late jobs.

Theorem: EDD is optimal for $1|r_j, pmtn|T_{max}$

Proof: Exchanging argument.

Minimizing Tardiness with Release Dates and No Preemptions.

Theorem: $1|r_j|T_{max}$ is NP-hard.

Proof: A reduction from Partition.

The partition problem:

Input: a set of n numbers, $A =\{a_1, a_2,\ldots, a_n\}$, such that $\sum_{j\in A} a_j = 2B$.

Output: Is there a subset S' of A such that $\sum_{j\in A'} a_j= B$?

Example: A={5, 5, 7,3, 1, 9, 10}; B=20

A possible partition: A'={10,5,5}, A-A'={7,3,1,9}

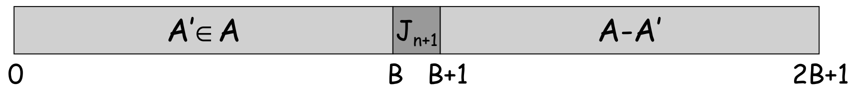
Minimizing Tardiness with Release Dates and No Preemptions.

Hardness proof for $1|r_j|T_{max}$:

Given an instance for partition, $A =\{a_1, a_2,\ldots, a_n\}$, s.t. $\sum_j a_j = 2B$, we build an instance for $1|r_j|T_{max}$ such that $T_{max}=0$ if and only if A has a partition:

For each item, $a_j \in A$, we have a job j with $p_j = a_j$, $r_j=0$, and $d_j = 2B+1$. In addition, we have the job, J_{n+1}, with $p_{n+1}=1$, $r_{n+1}= B$, and $d_{n+1}=B+1$.

- To achieve $T_{n+1}=0$, J_{n+1} must be scheduled in [B,B+1]



- The schedule of J_{n+1} induces a partition

Moore's Algorithm for $1||\sum_j U_j$

The objective: minimize the number of late jobs.

U_j is the lateness indicator (=0 if $C_j\leq d_j$; 1 if $C_j > d_j$).

The problem: $1||\sum_j U_j$

An optimal algorithm (Moore):

1. Order the jobs according to EDD rule (into A^*). The set R^* is empty.
2. If no job in A^* is late. A^*R^* is an optimal order.
3. Else, let k be the first job to be late in A^*.
4. Move to R^* the longest job among the first k jobs in A^*.
5. Update the completion times of jobs in A^*. Go to step 2.

Moore's Algorithm for $1||\sum_j U_j$ (example)

1. Order the jobs according to EDD rule (into A^*). The set R^* is empty.

j	p_j	d_j
1	1	2
2	5	7
3	3	8
4	7	11
5	9	13

A^*= {1-2-3-4-5} , R^* = { }.

2. According to this order, J_3 is the first to be late (C_3=9).

3. The longest job among the first three is J_2.

4. We move J_2 to R^*.

A^*= {1-3-4-5}, R^* = {2}

Moore's Algorithm for $1||\sum_j U_j$ (cont')

j	p_j	d_j
1	1	2
3	3	8
4	7	11
5	9	13

A^*= {1-3-4-5}, R^* = {2}

2. According to this order, J_5 is to first to be late (C_5=20).

3. The longest job among the first four (J_5 is the 4th in A^*) is J_5.

4. We move J_5 to R^*.

A^*= {1-3-4}, R^* = {2, 5}

j	p_j	d_j
1	1	2
3	3	8
4	7	11

Now, no job in A^* is late.

A^*R^*= 1-3-4-2-5 is optimal

$\sum_j U_j=2$

Scheduling with Cost-Functions

Each job,j, is associated with a cost function
$f_j = f_j(C_j)$
Examples: $f_1(C_1)= C_1^2 +1$; $f_2(C_2)= \log C_2$.
For each j, $f(C_j)$ is non-negative, and non-decreasing.

The problem: $1||f_{max}$.

Example 1: if $\forall j$, $f_j(x)=x$ then the problem $1||f_{max}$ is to minimize the makespan ($1||C_{max}$).

Example 2: $\forall j$, $f_j(C_j)=w_jT_j$. Now $1||f_{max}$ is the problem of minimizing the weighted tardiness.

The Least-Cost-Last (LCL) Algorithm

The LCL algorithm determines the processing order of the jobs from the last-to-process job to the first-to-process one.

At each stage, the last-to-process job among the remaining ones is the job whose schedule as last causes the smallest cost.

The time complexity of LCL is $O(n^2)$: there are n candidates in the first iteration, n-1 in the second iteration, and so on (assuming that for each j,x, the value of $f_j(x)$ can be computed in O(1)).

The Least-Cost-Last (LCL) Algorithm

Example: $\forall j$, $f_j(C_j)=w_jT_j$. (minimizing the maximal weighted tardiness).

j	p_j	d_j	w_j	C_j if last	w_jT_j if last
1	3	4	1	7	$1\cdot3=3$ ← minimal
2	2	6	8	7	$8\cdot1=8$
3	2	3	3	7	$3\cdot4=12$

➔ J_1 is last (third)

j	p_j	d_j	w_j	C_j if last	w_jT_j if last
2	2	6	8	4	0 ← minimal
3	2	3	3	4	$3\cdot1=3$

➔ J_2 is second

➔ The optimal schedule is $\{J_3, J_2, J_1\}$. Max $w_jT_j=3$

The Least-Cost-Last (LCL) Algorithm

Theorem: LCL is optimal for $1||f_{max}$.

Proof: Let J be the set of jobs. Let $f^*_{max}(S)$ be the value of an optimal solution for a set S of jobs. Let $b_1 = \min_j f_j(\sum_j p_j)$ and Let $b_2 = \max_j f^*_{max}(J-\{j\})$.

Claim 1: Each of b_1, b_2 is a lower bound for $f^*_{max}(J)$.

Proof: b_1 is a lower bound since some job must be last and have $C_j = \sum_j p_j$. b_2 is a lower bound since f is non-negative and $\forall j$, $f^*_{max}(J) \geq f^*_{max}(J-\{j\})$. ■

Claim 2: LCL achieves $f_{max} = \max\{b_1,b_2\}$.

Proof: Homework (by induction on n).

Scheduling with Cost-Functions

The problem $1|prec|f_{max}$

In scheduling problems with precedence constraints, we are given a directed precedence graph. An edge from i to j implies that we can start process J_j only after J_i is completed.

Theorem: LCL is optimal for $1|prec|f_{max}$.

In the implementation of LCL for instances with precedence constraints, the only candidates for the last position are jobs that no other jobs depend on them.

Flow-shop Scheduling

In a flow-shop schedule with m machines, $M_1,M_2,\ldots,M_m$, all the jobs must be processed by all the machines in the same order (which is, w.l.o.g., $M_1,M_2,\ldots,M_m$). For each job j and machine i, $p_{j,i}$ is the processing time required by J_j on M_i.

Example:
Two machines,
three jobs.

	pizza	pie	cake
chef	8	10	4
oven	5	20	30

Flow-shop Scheduling

- The problem $Fm||C_{max}$ is NP-hard for any m > 2.
- We will see a simple optimal algorithm for m=2 (Johnson 1954).

Observations for F2:

- In any F2-schedule, the machine M_2 is idle first, then it processes jobs, then it may be idle again, process again, and so on, depending on the flow of jobs from M_1.
- M_1 is never idle (or idles can be removed).
- Since all jobs are available at time t=0, our goal is to reduce the time in which M_2 is idle, waiting for the job currently processed by M_1.

Flow-shop Scheduling on Two Machines

Definition: A permutation schedule is a schedule in which the jobs are processed in the same order by M_1 and M_2.

Lemma: There exists an optimal schedule which is a permutation schedule.

Proof idea: if J_j precedes J_k on M_1, then J_j is available to M_2 before J_k and so, if J_j precedes J_k on M_2 we can swap their processing on M_2 without hurting the makespan.

$$\begin{tikzcd} J_k\ J_j \arrow[r] & \boxed{M_1} \arrow[r] & J_k\ J_j \arrow[r] & \boxed{M_2} \end{tikzcd}$$

Flow-shop Scheduling on Two Machines

Let A be the set of jobs j for which $p_{j,1} \le p_{j,2}$.

Let B be the set of jobs j for which $p_{j,1} > p_{j,2}$.

Johnson Rule: Sort the jobs in the following way: first the jobs of A in non-decreasing order of p_{j1}, then the jobs of B in non-increasing order of p_{j2}. Schedule the jobs on the two machines according to this order.

Example:

A = {pie, cake}

B = {pizza}

	pizza	pie	cake
chef	8	10	4
oven	5	20	30

Optimal order = {cake, pie, pizza}

Optimality of Johnson Rule for $F2||C_{max}$

- For a given permutation schedule, number the jobs according to the order they are scheduled.
- Let J_k be the first job on M_2 after its last idle section. J_k is not waiting between M_1 and M_2.
- $C_k = p_{1,1}+p_{1,2}+\ldots+p_{1,k}+p_{2,k}$.
- M_2 is not idle when the rest of the jobs are processed, thus, $C_{max}= p_{1,1}+\ldots+p_{1,k}+p_{2,k}+p_{2,(k+1)}+\ldots+p_{2,n}$.

➔ The makespan is determined by n+1 values.

➔ For any c, we can reduce c from all the p_{ij} values, without changing the relative performance of different permutation schedules.

Optimality of Johnson Rule for $F2||C_{max}$

Theorem: Johnson rule is optimal for $F2||C_{max}$.

Proof: By induction on the number of jobs, n.

Base: For n=1, any schedule with no idle is optimal.

Step: Assume that Johnson rule is optimal for n-1 jobs, and consider an instance with n jobs.

Let $c = \min_j\{\min\{p_{1,j}, p_{2,j}\}\}$. Reduce c from all the $p_{j,i}$ values. As a result, there exists a job, with $p_{1,j}=0$ or $p_{2,j}=0$. If $p_{1,j}=0$ then $j \in A$ and it is first in the Johnson-order of A. If $q_{2,j}=0$ then $j \in B$ and it is last in the Johnson-order of B.

Optimality of Johnson Rule for $F2||C_{max}$

If $p_{1j}=0$, then there exist an optimal schedule in which j is first (and can be processed by M_2 with no delay), and if $p_{2j}=0$, then there exists an optimal schedule in which j is last (and do not cause any delay to the makespan of M_2.

By the induction hypothesis, Johnson rule is optimal for J-{j}. By the above, Johnson rule places j optimally. ■