

# Algorithm Design

## Scheduling Algorithms

### Part 2

Parallel machines.  
Open-shop Scheduling.  
Job-shop Scheduling.

# Parallel Machines

- $n$  jobs need to be scheduled on  $m$  machines,  $M_1, M_2, \dots, M_m$ .
- Each machine can process at most one job at any time.
- Each job can be processed by at most one machine at any time.
- **Identical machines** (denoted P) - all the machines have the same rate. Processing time of job  $j = p_j$
- **Uniform machines** (denoted Q) - each machine has a rate,  $s_i$ , uniform for all the jobs processed on it. Processing time of job  $j = p_j / s_i$ .
- **Unrelated machines** (denoted R) - Specific values of  $p_{i,j}$ .  $p_{i,j}$  = The processing time of job  $j$  on machine  $i$ .

# Parallel Machines, the problem $P|| \sum_j C_j$

**Theorem:** SPT is optimal for  $P|| \sum_j C_j$

**Proof:** Assume  $n=zm$  (w.l.o.g we can add jobs with  $p_j=0$ , why?)

Index the jobs such that  $p_1 \leq p_2 \leq \dots \leq p_n$ .

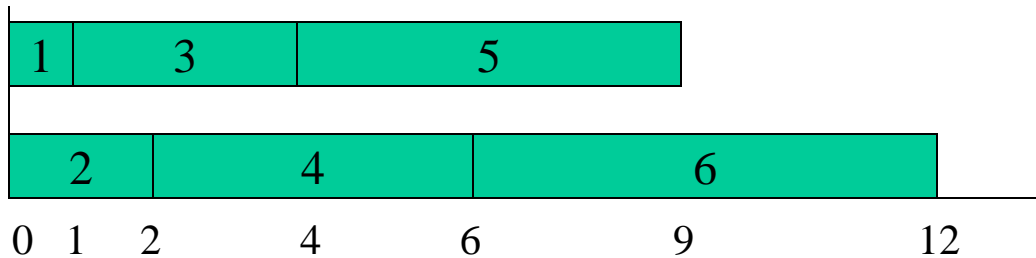
In SPT the  $m$  jobs  $J_{(k-1)m+1}, \dots, J_{km}$ , are scheduled in the  $k^{\text{th}}$  locations on each of the  $m$  machines. Their processing time is 'counted'  $z+1-k$  times in  $\sum_j C_j$ .

**Note:** We can switch jobs in the  $k^{\text{th}}$  location on different machines without affecting  $\sum_j C_j$

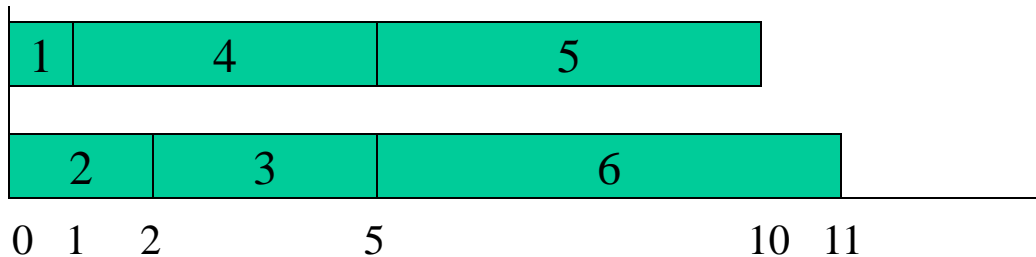
## SPT for $P|| \sum_j C_j$ . Example.

$m=2$ ,

6 jobs with processing times 1,2,3,4,5,6



$$\sum_j C_j = 34$$



$$\sum_j C_j = 34$$

# Parallel Machines, the problem $P|| \sum_j C_j$

$\sum_j C_j$  is the result of the following vector multiplication:

$$\underbrace{(z, \dots, z)}_{m \text{ times}}, \underbrace{(z-1, \dots, z-1)}_{m \text{ times}}, \underbrace{(1, \dots, 1)}_{m \text{ times}} (p_1, p_2, \dots, p_n)$$

The left vector is non-increasing, so the multiplication value is minimized if the other vector is non-decreasing, as implied by SPT.

# The problem $P|| \sum_j w_j C_j$

This problem is NP-hard.

It can be solved using **dynamic programming** or **branch and bound**.

In any optimal solution, the jobs scheduled on one machine are scheduled according to WSPT rule ( $p_1/w_1 \leq p_2/w_2 \leq \dots$ ) otherwise exchanges can improve the objective function.

**However**, the problem of partitioning the jobs among the machines is NP-hard.

# The problem $P||C_{\max}$

**Theorem:** The problem  $P||C_{\max}$  is NP-hard

**Proof:** Reduction from **Partition**.

**Reminder:** The partition problem:

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

**Output:** Is there a subset  $A'$  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:**  $A' = \{10, 5, 5\}$ ,  $A - A' = \{7, 3, 1, 9\}$

# The problem $P||C_{\max}$

Reduction from Partition to  $P||C_{\max}$  :

Given an instance for partition,  $A = \{a_1, a_2, \dots, a_n\}$   
such that  $\sum_{j \in A} a_j = 2B$ .

Build an instance for  $P2||C_{\max}$  such that the  
makespan is  $B$  if and only if there is a  
partition.

There are  $n$  jobs, the processing time of  $J_i$  is  $a_i$ .

If there is a schedule with makespan  $= B$  the jobs  
scheduled on  $M_1$  corresponds to items in  $S'$  -  
their total size must be  $B$ .

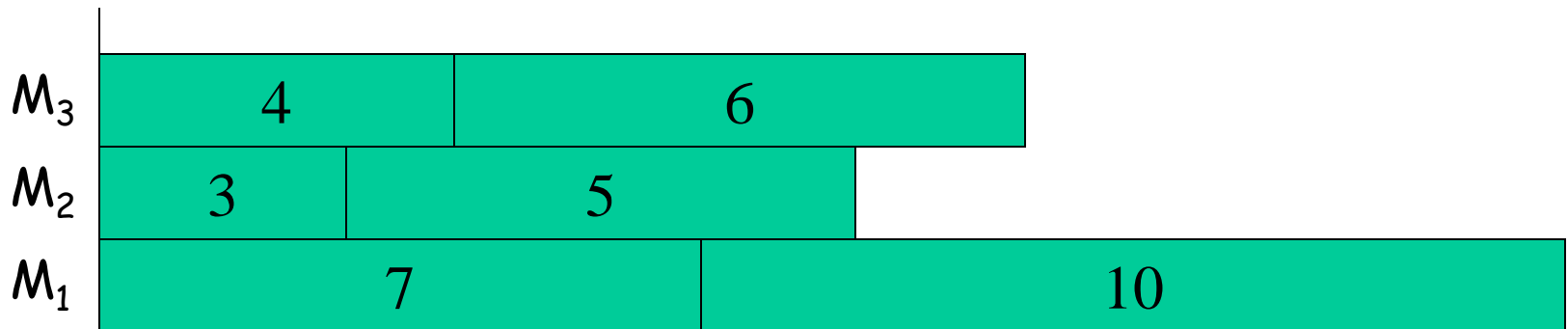


# The problem $P||C_{\max}$

**List Scheduling** [Graham 1966]:

A greedy algorithm: always schedule a job on the least loaded machine.

Example:  $m=3$   $\sigma = 7 \ 3 \ 4 \ 5 \ 6 \ 10$



Makespan = 17

# List Scheduling for $P||C_{\max}$

**Theorem:** List Scheduling provides a  $(2 - \frac{1}{m})$ -approximation for the problem  $P||C_{\max}$ .

**Proof:** Let  $H_j$  denote the last completion time on the  $j^{\text{th}}$  machine. Let  $k$  be the job that finishes last and determines  $C_{LS}$ .

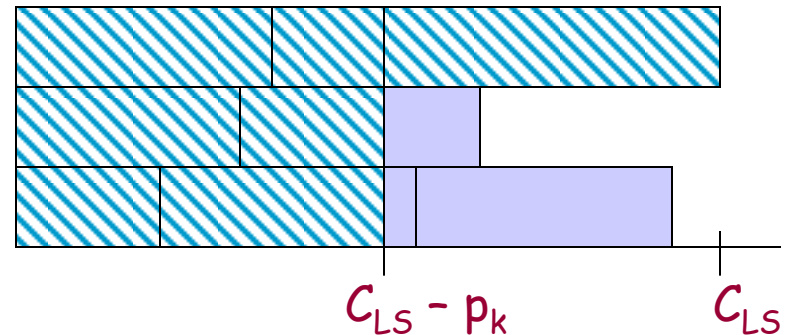
All the machines are busy when  $k$  starts its processing, thus,  $\forall j, H_j \geq C_{LS} - p_k$ .

For at least one machine (that processes  $k$ )  $H_j = C_{LS}$ .

$$\rightarrow \sum_i p_i = \sum_j H_j \geq (m-1)(C_{LS} - p_k) + C_{LS}.$$

$$\rightarrow \sum_i p_i + (m-1)p_k \geq mC_{LS}.$$

$$\rightarrow C_{LS} \leq \frac{1}{m} \sum_i p_i + p_k \frac{(m-1)}{m}.$$



# List Scheduling for $P||C_{\max}$

$$\rightarrow C_{LS} \leq 1/m \sum_i p_i + p_k (m-1)/m.$$

Consider an **optimal** schedule.

$C_{\text{opt}} \geq \max_i p_i \geq p_k$  (some machine must process the longest job).

$C_{\text{opt}} \geq 1/m \sum_i p_i$  (if the load is perfectly balanced).

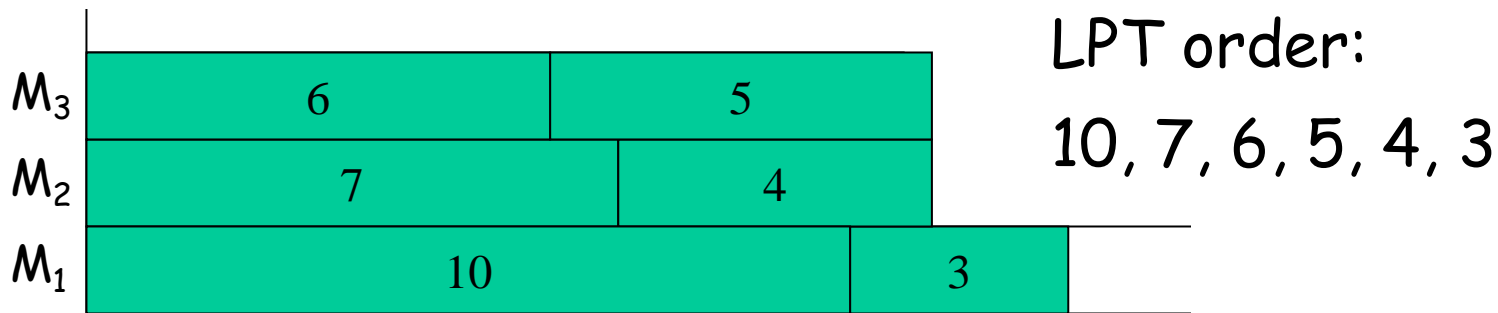
Therefore,

$$C_{LS} \leq C_{\text{opt}} + C_{\text{opt}} (m-1)/m = (2-1/m) C_{\text{opt}}.$$

# Longest Processing Time Rule

The  $(2-1/m)$ -ratio is for arbitrary order of the jobs.  
If the jobs are known in advance (offline problem)  
it is possible to determine the assignment order.

**LPT algorithm:** List scheduling where the jobs are  
arranged in non-increasing order of  $p_1 \geq p_2 \geq \dots \geq p_n$ .



Makespan = 13

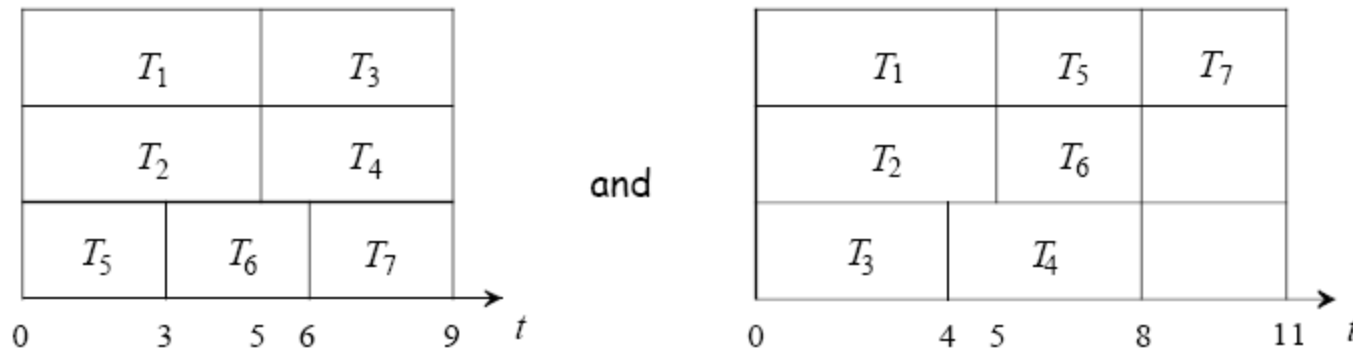
# Longest Processing Time (LPT) Rule

**Theorem:** LPT provides a  $(\frac{4}{3} - \frac{1}{3m})$  - approximation for the problem  $P||C_{\max}$ .

**Proof:** In class.

**This analysis is tight:** Consider  $n = 2m + 1$  jobs,  
 $p = [2m - 1, 2m - 1, 2m - 2, 2m - 2, \dots, m + 1, m + 1, m, m, m]$ .

An optimal schedule and an LPT schedule are ( $m=3$ ):



## The problem $P|pmtn|C_{\max}$

When preemptions are allowed, the problem is optimally solvable in poly-time.

We have two lower bounds for  $opt$ :

$C_{opt} \geq \max_i p_i$  (some machine must process the longest job).

$C_{opt} \geq 1/m \sum_i p_i$  (if the load is perfectly balanced).

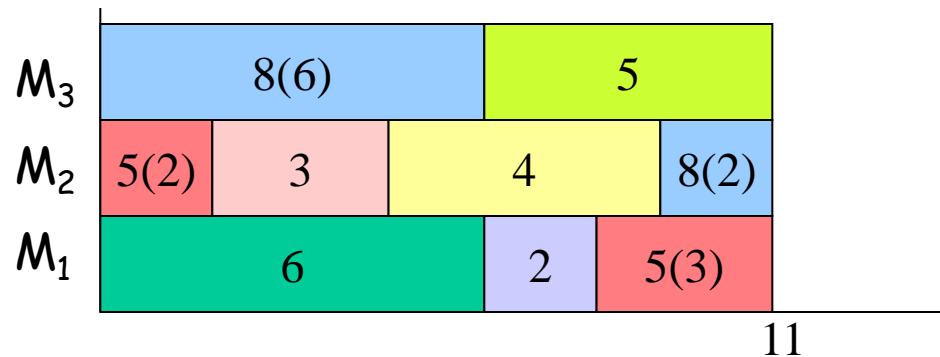
Let  $w = \max(\max_i p_i, 1/m \sum_i p_i)$

# An optimal algorithm for $P|pmtn|C_{\max}$

1. Calculate  $w = \max(\max_i p_i, 1/m \sum_i p_i)$
2. Consider the jobs in arbitrary order, schedule the jobs one after the other on the machines. Move to  $M_{i+1}$  after  $M_i$  allocated  $w$  processing units (maybe preempt the last job on  $M_i$ ).

**Example:**  $m=3$ , Jobs lengths are 6,2,5,3,4,8,5

$$w = \max(7, 11) = 11$$



# The problem $P|\text{prec}, p_j=1|C_{\max}$

**prec** - There are precedence constraints.

Given as an acyclic graph.

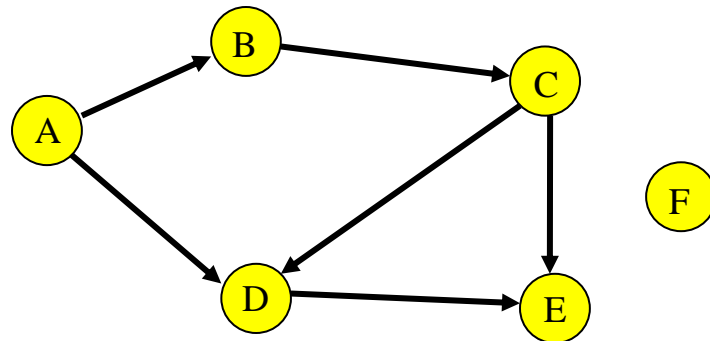
On a single machine -

any topological sort can do.  $C_{\max} = n$

**Example:**

Possible topo-sort and schedule: **A-B-F-C-D-E**

$C_{\max} = 6$



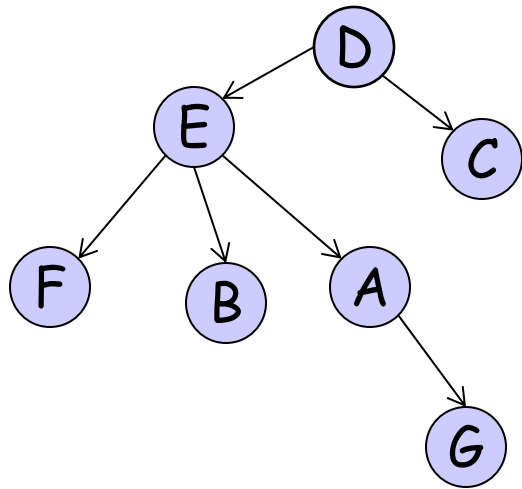
The problem is still simple if jobs have different lengths. However, on parallel machines the problem is NP-hard even for unit-length jobs.



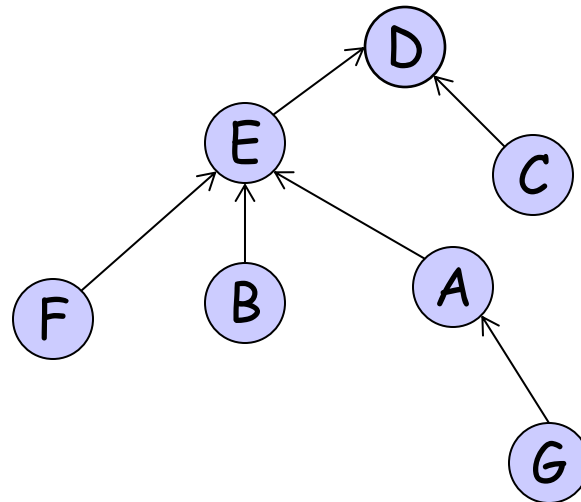
$$P|tree, p_j=1|C_{\max}$$

The precedence constraints graph is a tree.

There are two special cases:



**Out-tree:** each job has at most one predecessor (in-degree at most 1).



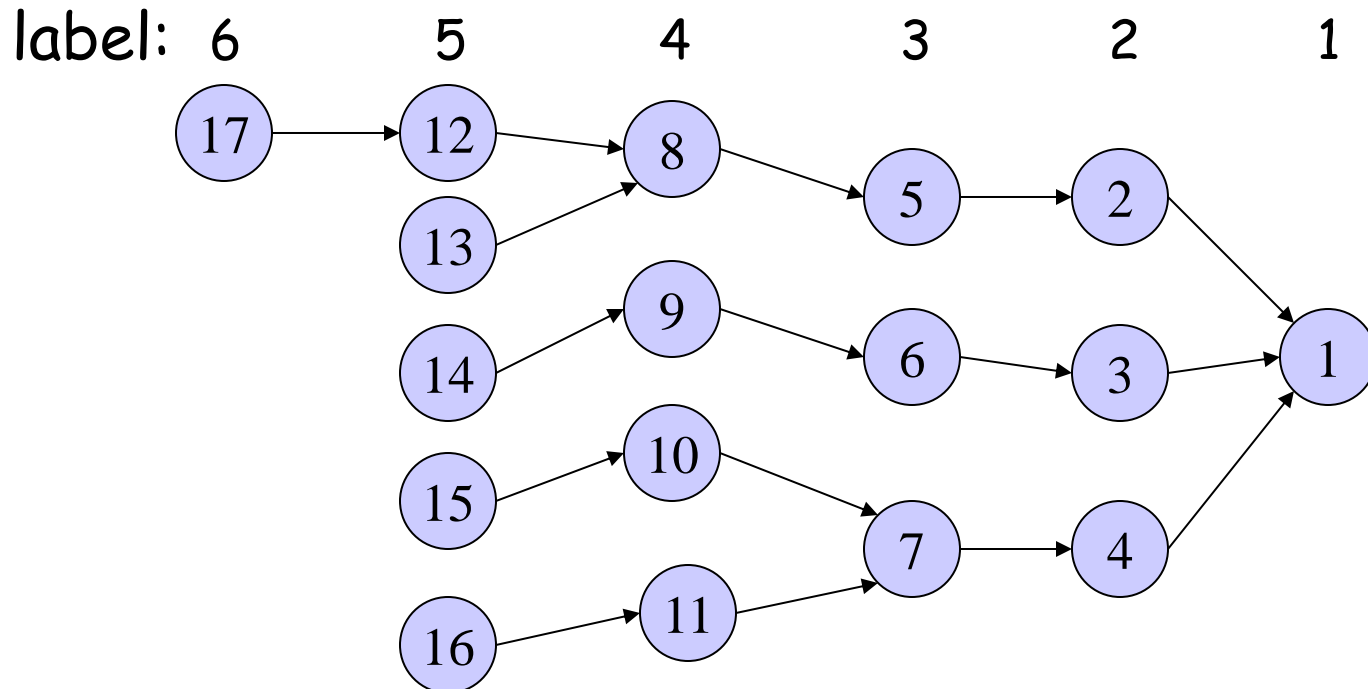
**In-tree:** each job has at most one consecutive (out-degree at most 1).

$$P|in-tree, p_j=1|C_{\max}$$

## Hu's Algorithm for in-tree.

### Phase 1: Labeling

1. Label with 1 each 'sink job' (with out-degree=0)
2. For every labeled job  $j$ , find the jobs that immediately precede  $j$  and label each of them with  $\text{label}(j)+1$ .



$P|in-tree, p_j=1|C_{max}$

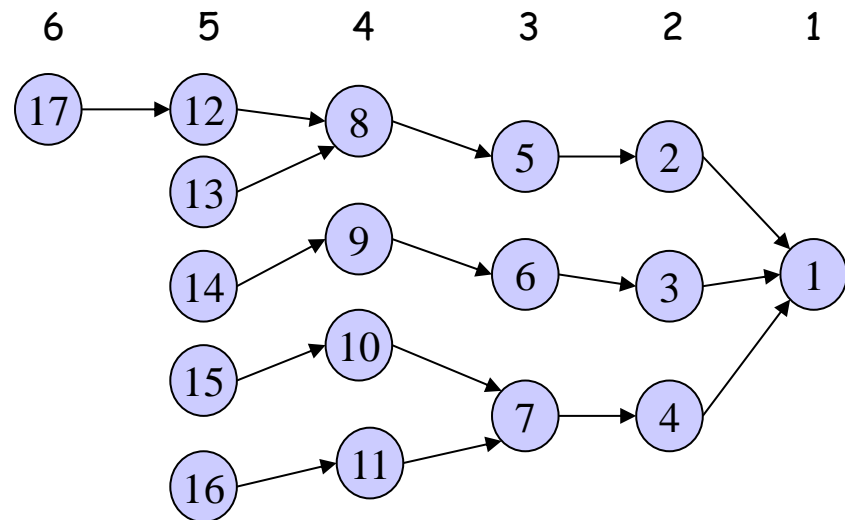
## Hu's Algorithm for in-tree.

### Phase 2: scheduling

1. If the number of source-jobs is at most  $m$ , schedule them and leave the non-used machines idle.
2. Otherwise, schedule the  $m$  source-jobs with the largest labels.
3. Remove the scheduled jobs from the instance and return to step 1.

Example:  $m=3$ .

$M_3$	17
$M_2$	15
$M_1$	16



$P|in-tree, p_j=1|C_{max}$

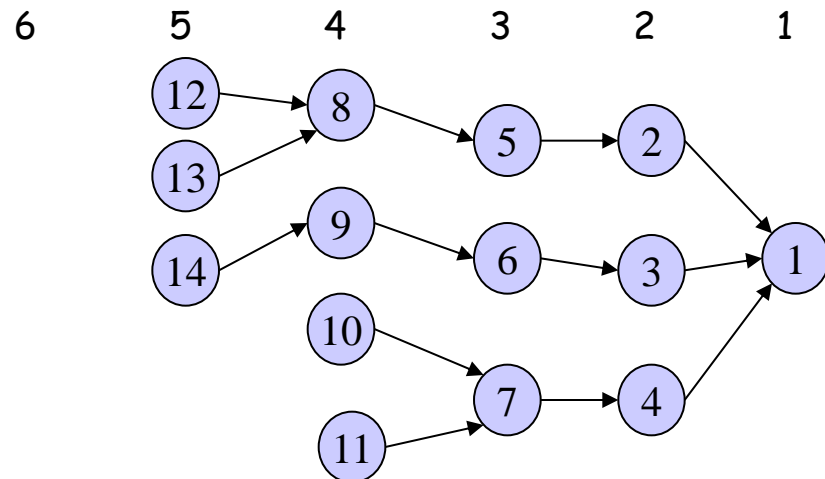
## Hu's Algorithm for in-tree.

### Phase 2: scheduling

1. If the number of source-jobs is at most  $m$ , schedule them and leave the non-used machines idle.
2. Otherwise, schedule the  $m$  source-jobs with the largest labels.
3. Remove the scheduled jobs from the instance and return to step 1.

Example:  $m=3$ .

$M_3$	17	12
$M_2$	15	13
$M_1$	16	14



$P|in-tree, p_j=1|C_{max}$

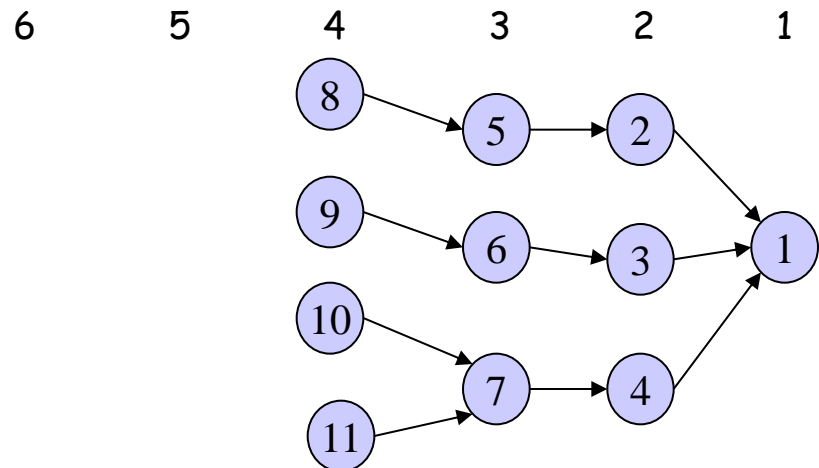
## Hu's Algorithm for in-tree.

### Phase 2: scheduling

1. If the number of source-jobs is at most  $m$ , schedule them and leave the non-used machines idle.
2. Otherwise, schedule the  $m$  source-jobs with the largest labels.
3. Remove the scheduled jobs from the instance and return to step 1.

Example:  $m=3$ .

$M_3$	17	12	8
$M_2$	15	13	9
$M_1$	16	14	10



$P|in-tree, p_j=1|C_{max}$

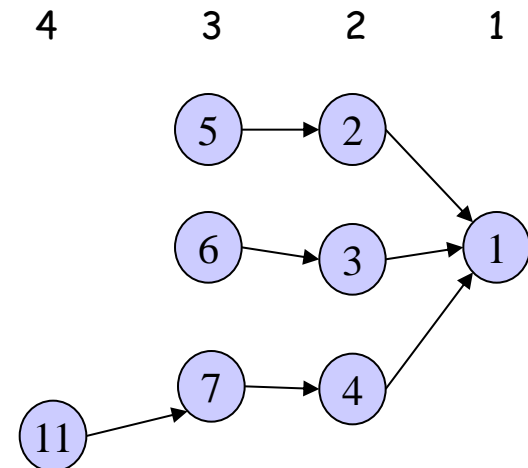
## Hu's Algorithm for in-tree.

### Phase 2: scheduling

1. If the number of source-jobs is at most  $m$ , schedule them and leave the non-used machines idle.
2. Otherwise, schedule the  $m$  source-jobs with the largest labels.
3. Remove the scheduled jobs from the instance and return to step 1.

Example:  $m=3$ .

$M_3$	17	12	8	5
$M_2$	15	13	9	6
$M_1$	16	14	10	11



$$P|in-tree, p_j=1|C_{\max}$$

## Hu's Algorithm for in-tree.

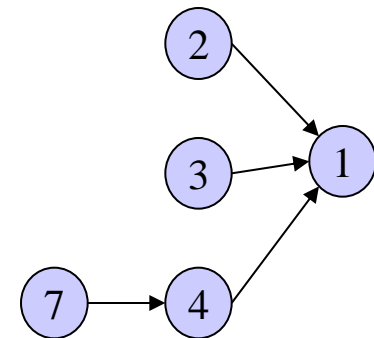
### Phase 2: scheduling

1. If the number of source-jobs is at most  $m$ , schedule them and leave the non-used machines idle.
2. Otherwise, schedule the  $m$  source-jobs with the largest labels.
3. Remove the scheduled jobs from the instance and return to step 1.

6      5      4      3      2      1

Example:  $m=3$ .

$M_3$	17	12	8	5	2		
$M_2$	15	13	9	6	3		
$M_1$	16	14	10	11	7	4	1



# $P|out-tree, p_j=1|C_{\max}$

## Phase 1: labeling

Label(j)=number of jobs waiting for j.

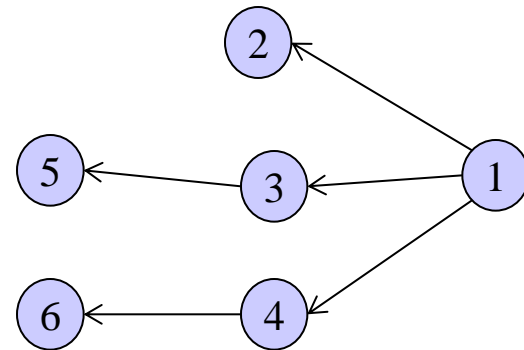
## Phase 2: scheduling

same as in in-tree.

j	1	2	3	4	5	6
Label(j)	5	0	1	1	0	0

Example:  $m=2$ .

$M_2$		3	2	
$M_1$	1	4	5	6





## More on $P|p_j=1|C_{\max}$

**Scheduling forests:** A forest consisting of in-trees can be scheduled by adding a dummy task that is an immediate successor of only the roots of in-trees, and then by applying Hu's Algorithm.

**Scheduling out-forests:** A schedule for an out-tree can be constructed by changing the orientation of edges, applying Hu's Algorithm to the obtained intree and then reading the schedule backwards, i.e. from right to left.

**Remark:** The problem of scheduling opposing forests (that is, combinations of in-trees and out-trees) on an arbitrary number of processors is NP-hard.

# The Problem $P|prec|C_{\max}$

An NP-hard problem ( $P||C_{\max}$  is already NP-hard)

**List scheduling algorithm:** Schedule the jobs greedily according to some topological sort.

LS have unexpected behavior: the schedule length may increase if:

- the number of processors increases,
- task processing times decrease,
- precedence constraints are weakened, or
- the priority list changes.

# The Problem $P|prec|C_{\max}$

**Given:** Job set  $T$  with

- vector of processing times  $p$
- precedence constraints  $\theta$
- priority list  $L$
- $m$  identical processors

Let  $C_{\max}$  be the length of the list schedule.

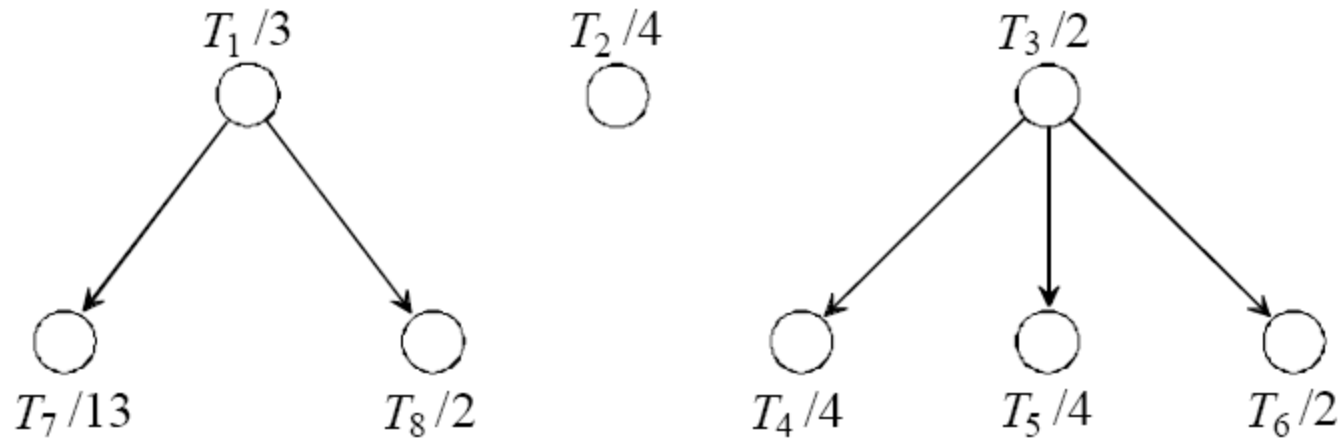
**On the other hand, let the above parameters be changed:**

- vector of processing times  $p' \leq p$  (component-wise),
- relaxed precedence constraints  $\theta' \subseteq \theta$ ,
- priority list  $L'$
- and another number of processors  $m'$

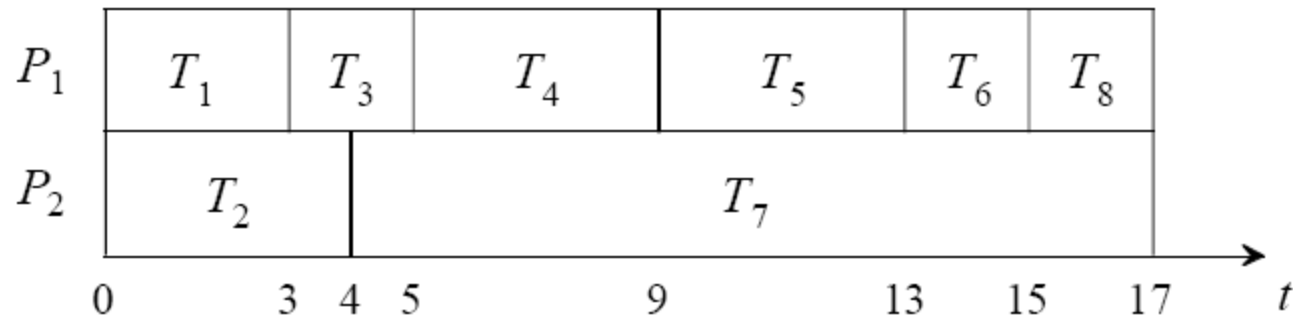
Let the new value of schedule length be  $C'_{\max}$ .

# The Problem $P|prec|C_{\max}$

(a)



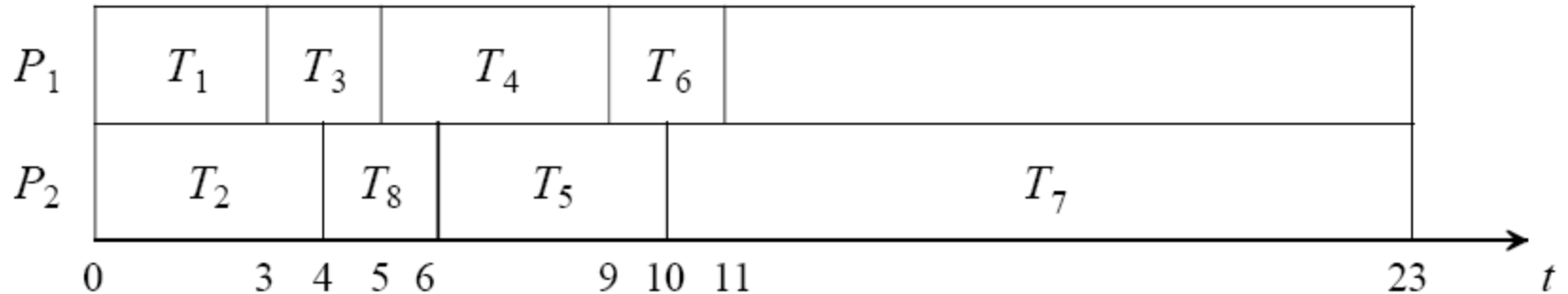
(b)



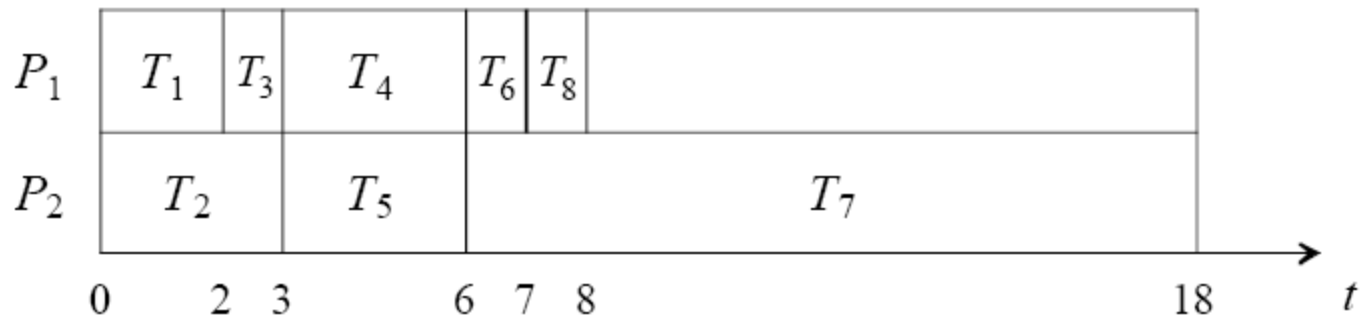
(a) A task set,  $m = 2$ ,  $L = (T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8)$ ,

(b) an optimal schedule

# The Problem $P|prec|C_{\max}$



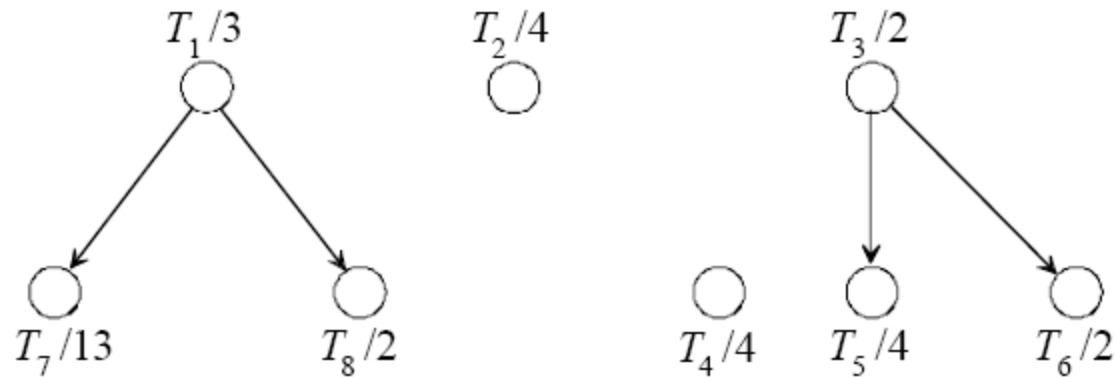
A new list  $L' = (T_1, T_2, T_3, T_4, T_5, T_6, T_8, T_7)$ .



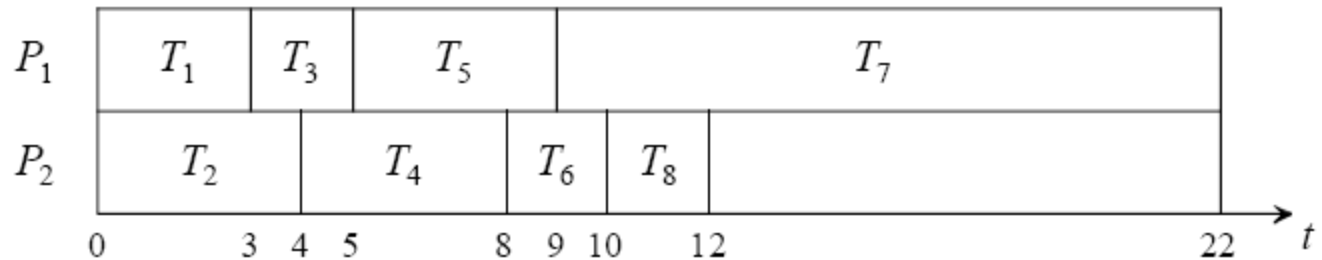
Processing times decreased;  $p'_j = p_j - 1, j = 1, 2, \dots, n$ .

# The Problem $P|prec|C_{\max}$

(a)

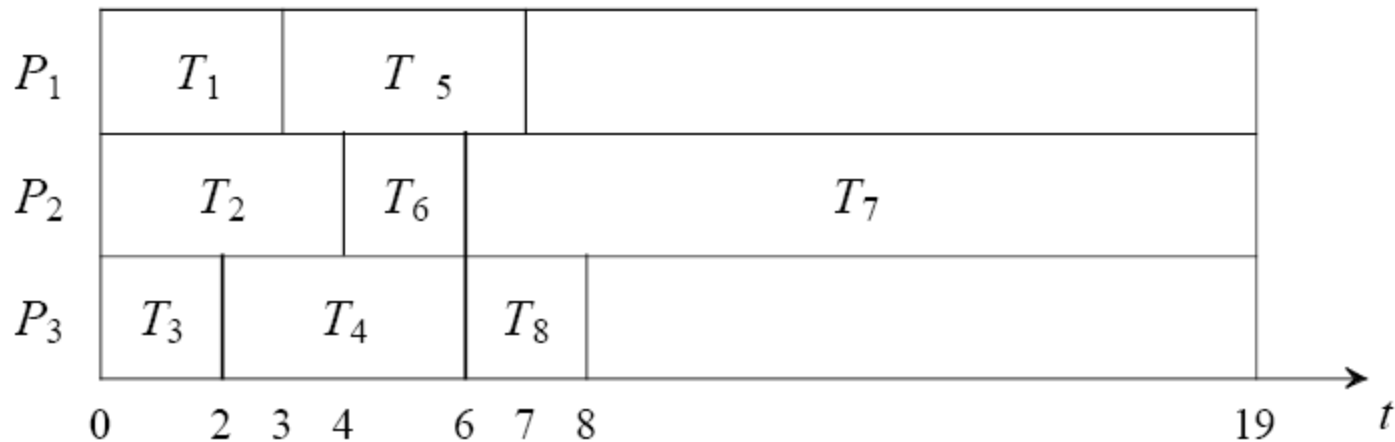


(b)



(a) Precedence constraints weakened, (b) resulting list schedule.

# The Problem $P|prec|C_{\max}$



*Number of processors increased,  $m = 3$*

The 'paradox' is limited:

**Theorem** [Gra66] *Under the above assumptions,*

$$\frac{C'_{\max}}{C_{\max}} \leq 1 + \frac{m-1}{m'}$$

# The Problem $R|| \sum_j C_j$

- **Unrelated machines:** for each machine  $i$  and job  $j$  the time  $p_{ij}$  is specified.
- For a single machine  $1|| \sum_j C_j$  is solvable by SPT rule.
- The problems  $P|| \sum_j C_j$  and  $Q|| \sum_j C_j$  are solvable by variants of SPT.
- For related machines, the processing times cannot be sorted.

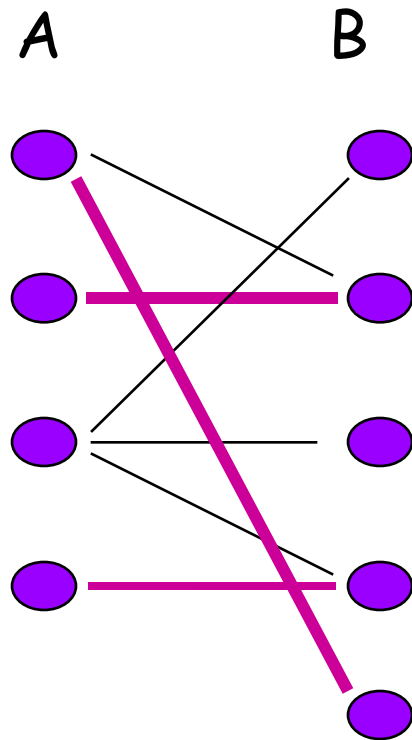
**Observation:** If  $J_j$  is the  $k^{\text{th}}$  from last job to run on a machine, it contributes exactly  $k$  times its processing time to the sum of completion times.



$k=3$ ,  $p_{ij}$  is counted  
3 times in  $\sum_j C_j$



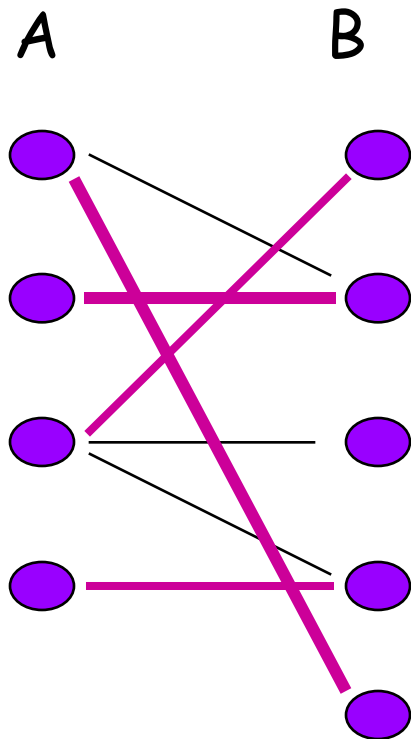
# Tool: Bipartite matching



Given a bipartite graph on two sets of vertices  $A$  and  $B$ , and an edge set  $E \subseteq A \times B$ , a **matching**  $M$  is a subset of the edges, such that each vertex in  $A$  and  $B$  is an endpoint of at most one edge of  $M$ .

The optimal algorithm for  $R || \sum_j C_j$  will match jobs to locations in the schedule.

# Tool: Bipartite Matching



A perfect matching:  $|M| = |A| \leq |B|$   
(w.l.o.g.  $|A| \leq |B|$ )

- It is possible to assign weights to the edges, and define the weight of a matching to be the sum of the weights of the matching edges.

When a perfect matching exists, it is possible to compute in polynomial time a **minimum weight perfect matching**.

# The Problem $R || \sum_j C_j$

Define a bipartite with  $V = A \cup B$  as follows:

$A$  represents the set of  $n$  jobs (a single node per job).

$B$  consists of  $nm$  nodes,  $w_{ik}$ , where vertex  $w_{ik}$  represents the  $k^{\text{th}}$  from last position on machine  $i$ , for  $i=1, \dots, m$  and  $k=1, \dots, n$ .

The edges set  $E$  contains an edge  $(v_j, w_{ik})$  for every node in  $A$  and every node in  $B$  (a complete bipartite).

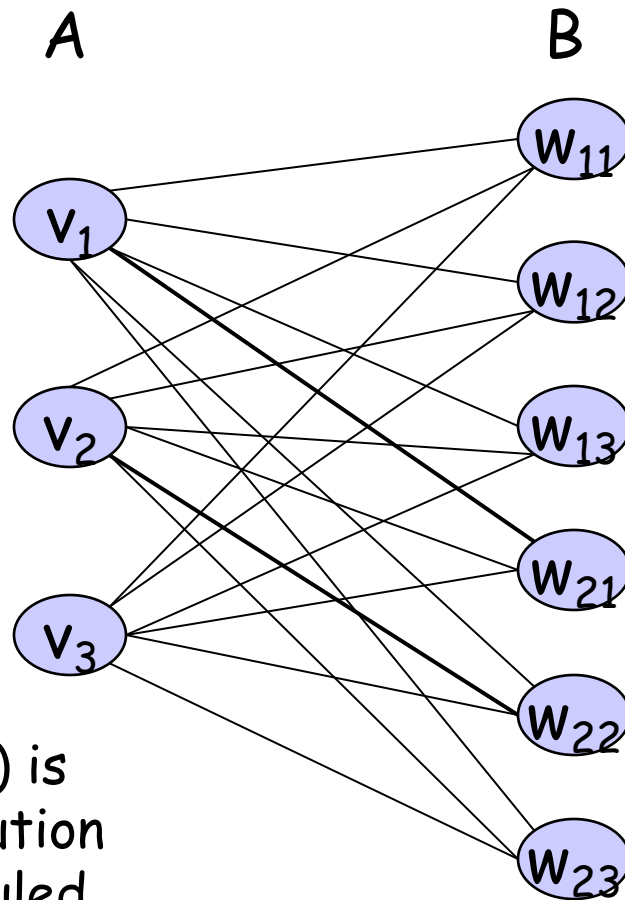
The weight of  $(v_j, w_{ik})$  is  $kp_{ij}$

# The Problem $R || \sum_j C_j$

Example:

$n=3, m=2$

$p_{ij}$	$J_1$	$J_2$	$J_3$
$M_1$	4	5	7
$M_2$	8	6	2



The weight of  $(v_1, w_{21})$  is 4 - the contribution of  $J_1$  to  $\sum_j C_j$  if scheduled as last on  $M_2$ .

← last position on  $M_2$

← before-last position on  $M_2$

The weight of  $(v_2, w_{22})$  is  $6 \cdot 2 = 12$  - the contribution of  $J_2$  to  $\sum_j C_j$  if scheduled as 2<sup>nd</sup> from last on  $M_2$ .

# The Problem $R||\sum_j C_j$

**Theorem:** a minimum weight perfect matching corresponds to an optimal schedule.

**Proof:** In class

**Note:** With identical machines we get (as expected) SPT.

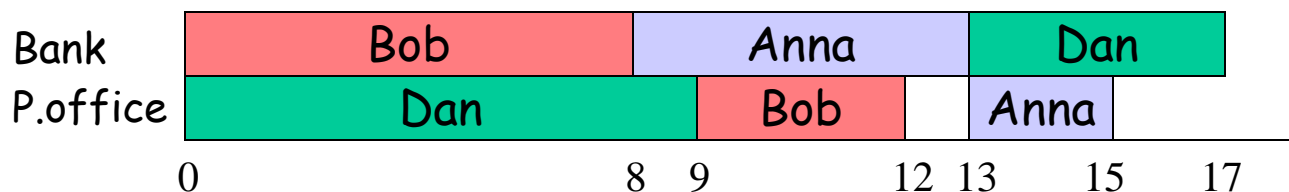
# Open-shop Scheduling

- Job  $J_j$  consists of  $m$  operations, to be processed non-preemptively on each of the  $m$  machines.
- The order in which the different operations are performed is not important.
- Two operations of the same jobs cannot be processed simultaneously.
- Example:

$m=2$

$n=3$

	Bob	Dan	Anna
Bank	8	4	5
Post-office	3	9	2



# The Problem $O||C_{\max}$

- The problem  $O||C_{\max}$  is NP-hard for any  $m > 2$ .
- We will see:
  1. A 2-approximation for any number of machines
  2. An optimal algorithm for  $O_2||C_{\max}$
- Two basic lower bounds:
  1.  $P_{\max}$  - maximum total processing time of a job (total length of operations composing this job)
  2.  $L_{\max}$  - maximum total processing time required from a single machine
- Clearly:  $C_{\max}(\text{opt})$  is at least  $\max(P_{\max}, L_{\max})$

	Bob	Dan	Anna
Bank	8	4	5
P.office	3	9	2

$$P_{\max} = 13 \text{ (Dan)}$$

$$L_{\max} = 17 \text{ (Bank)}$$

# The Problem $O||C_{\max}$

A **busy schedule** for  $O||C_{\max}$ : Whenever possible, schedule some operation of some job on a machine.

Can be implemented by a simple greedy algorithm.

**Theorem:** Any busy schedule is 2-approximation.

**Proof:** Consider the machine  $M'$  that finishes last. Let  $j'$  be the last job on  $M'$ . At any time during the schedule, either  $M'$  is processing a job, or  $j'$  is being processed by some other machine (why?).

The total time in which  $j'$  is processed is at most  $P_{\max}$

During the remaining time,  $M'$  is busy. But  $M'$  is busy at most  $L_{\max}$  time units.

Therefore  $C_{\max} = C_{j'} \leq P_{\max} + L_{\max} \leq 2 C_{\max}(\text{Opt})$



# Open-shop Scheduling

- In general,  $O||C_{\max}$  is NP-hard
- We will see an optimal schedule for  $O_2||C_{\max}$
- Denote  $p_{j1}=a_j, p_{j2}=b_j$  (Time to process  $j$  on  $M_1$  and  $M_2$  respectively)
- Let  $T_1=\sum_j a_j, T_2=\sum_j b_j$
- A lower bound for  $C_{\max}$  is  $\max(T_1, T_2, \max_j(a_j+b_j))$
- The algorithm achieves this lower bound.

# Optimal algorithm for $O_2 || C_{\max}$

Let  $A = \{j | a_j \geq b_j\}$  [need more time on  $M_1$ ]

Let  $B = \{j | a_j < b_j\}$  [need more time on  $M_2$ ]

Select a job  $J_r \in A$  such that  $a_r \geq \max \{b_j | j \in A\}$ .

Select a job  $J_l \in B$  such that  $b_l \geq \max \{a_j | j \in B\}$ .

Let  $A' = A - \{r\}$ ,  $B' = B - \{l\}$

Example:

	1	2	3	4	5	6	$T_j$
$a_j$	10	7	3	1	12	6	39
$b_j$	6	9	8	2	7	6	38

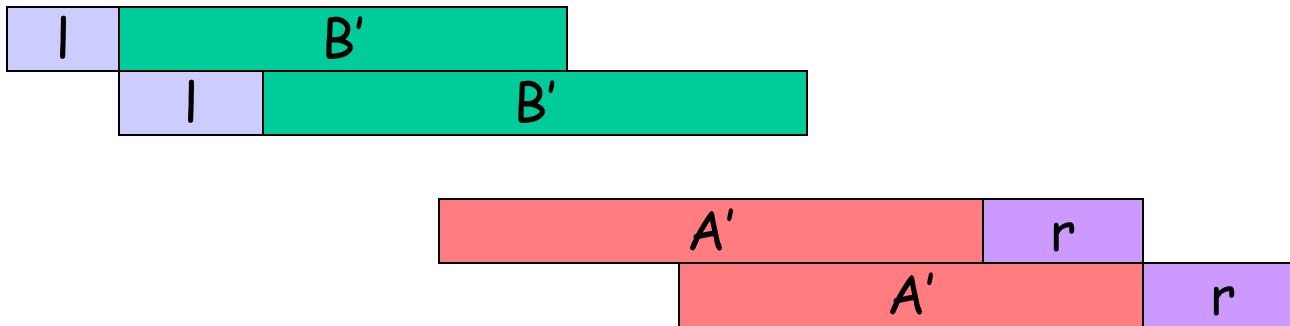
$A = \{1, 5, 6\}$ ,  $B = \{2, 3, 4\}$

$a_r \geq \max \{b_j | j \in A\}$  is true for  $r=1, 5$ . Select  $r=1$

$b_l \geq \max \{a_j | j \in B\}$  is true for  $l=2, 3$ . Select  $l=2$

# Optimal algorithm for $O_2 || C_{\max}$

Possible schedules for  $A' \cup \{r\}$  and  $B' \cup \{l\}$ :

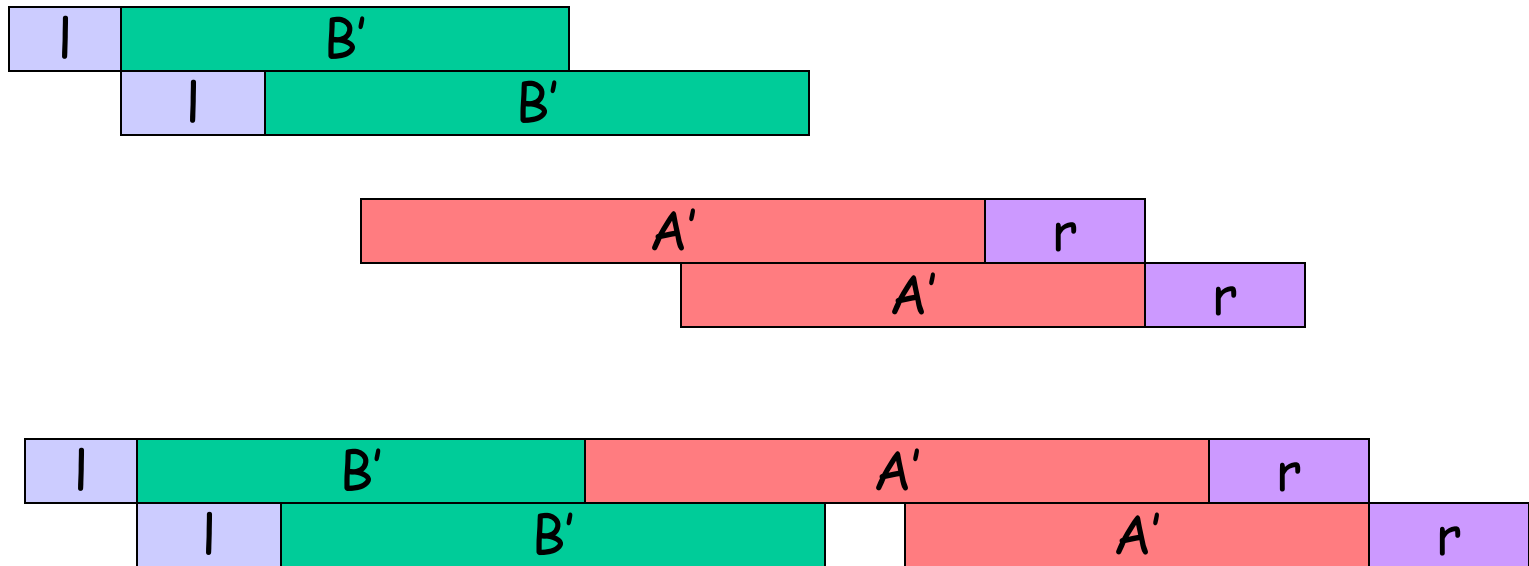


The jobs in  $A'$  and  $B'$  are in a fixed, arbitrary, order.

**Note:** There is no idle on any of the machines. (why?)

# Optimal algorithm for $O_2 || C_{\max}$

Let us 'glue' the two schedules.



Will meet on  $M_1$  if  $T_1 - a_l > T_2 - b_r$

# Optimal algorithm for $O_2 || C_{\max}$



'Slide' the jobs  $B'$  and  $J_1$  on  $M_2$  to the right.



A valid schedule (why?)

Move  $J_r$  to be first on  $M_2$



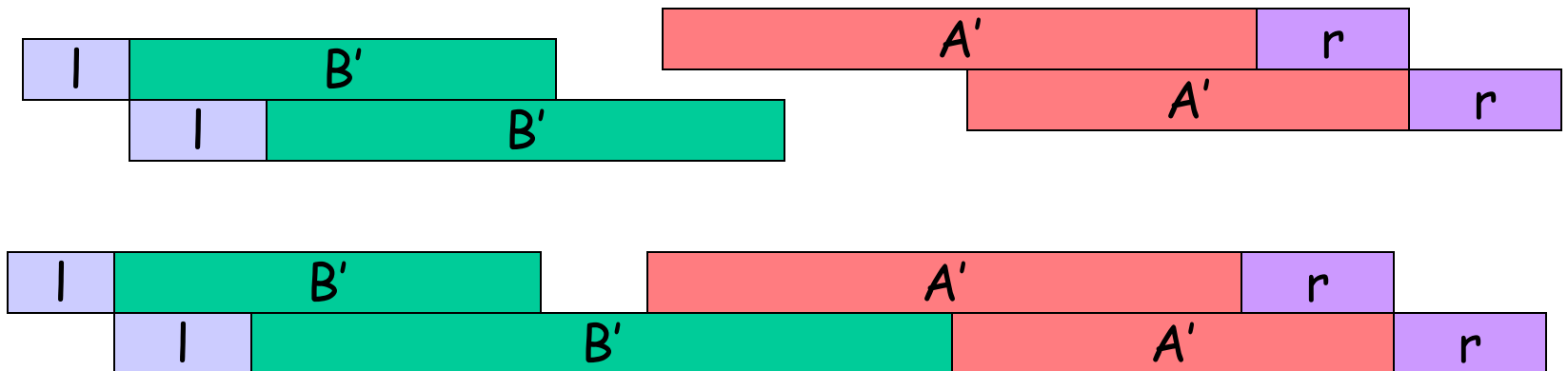
If  $a_r \leq T_2 - b_r$  The schedule length is  $\max(T_1, T_2)$

# Optimal algorithm for $O_2 || C_{\max}$



If  $a_r > T_2 - b_r$ , the schedule length is  $\max(T_1, a_r + b_r)$ .

If  $T_1 - a_l \leq T_2 - b_r$ , then when gluing, the schedules meet on  $M_2$  and the analysis is symmetric.



# Optimal algorithm for $O_2 || C_{\max}$

Let  $A = \{j | a_j \geq b_j\}$  [need more time on  $M_1$ ]

Let  $B = \{j | a_j < b_j\}$  [need more time on  $M_2$ ]

If  $A = \emptyset$  or  $B = \emptyset$  the problem is easy (why?)

Select a job  $J_r \in A$  such that  $a_r \geq \max \{b_j | j \in A\}$ .

Select a job  $J_l \in B$  such that  $b_l \geq \max \{a_j | j \in B\}$ .

Let  $A' = A - \{r\}$ ,  $B' = B - \{l\}$ .

If  $T_1 - a_l > T_2 - b_r$  then the optimal schedule is:

On  $M_1$ :  $(l, B', A', r)$ , on  $M_2$ :  $(r, l, B', A')$ .

Else, the optimal schedule is

On  $M_1$ :  $(B', A', r, l)$ , on  $M_2$ :  $(l, B', A', r)$ .

# Optimal algorithm for $O_2 || C_{\max}$

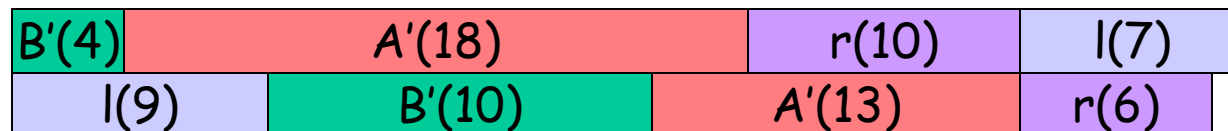
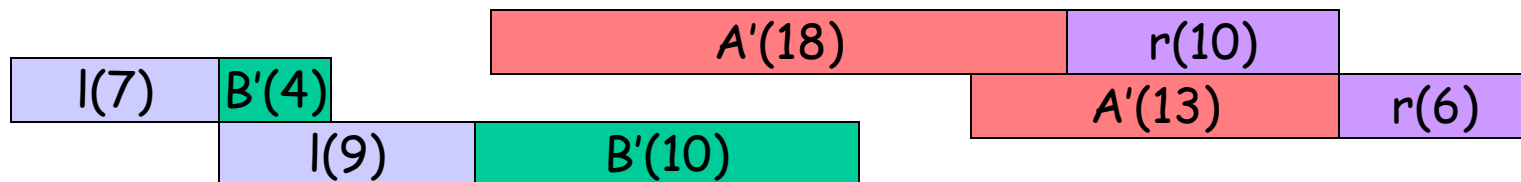
	1	2	3	4	5	6	$T_j$
1	10	7	3	1	12	6	39
2	6	9	8	2	7	6	38

$A=\{1,5,6\}$ ,  $B=\{2,3,4\}$

$a_r \geq \max \{b_j \mid j \in A\}$ . Select  $r=1$

$b_l \geq \max \{a_j \mid j \in B\}$ . Select  $l=2$

$A'=\{5,6\}$ ,  $B'=\{3,4\}$



$$C_{\max} = T_1 = 39$$



# Flow-shop Scheduling

In a flow-shop schedule with  $m$  machines,  $M_1, M_2, \dots, M_m$ , all the jobs must be processed by all the machines in the same order (which is, w.l.o.g.,  $M_1, M_2, \dots, M_m$ ). 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  $F_m || C_{\max}$  is NP-hard for any  $m > 2$ .
- We will see a simple optimal algorithm for  $m=2$  (Johnson 1954).

## Observations for $F_2$ :

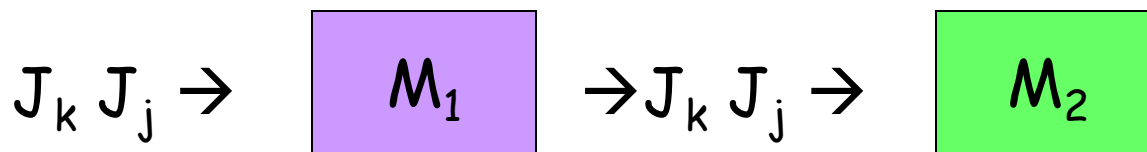
- In any  $F_2$ -schedule, the machine  $M_2$  is first idle, then it processes jobs, then it may be idle again, process again, and so on, depending on the flow of jobs from  $M_1$ .
- $M_1$  is never idle (or idles can be removed).
- Since all jobs are available at time  $t=0$ , our goal is to reduce 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  $J_k$  precedes  $J_j$  on  $M_2$  we can swap their processing on  $M_2$  without hurting the makespan.



# Flow-shop Scheduling on Two Machines

Denote  $p_{j1} = a_j, p_{j2} = b_j$ .

Let  $A$  be the set of jobs  $j$  for which  $a_j \leq b_j$ .

Let  $B$  be the set of jobs  $j$  for which  $a_j > b_j$ .

**Johnson Rule:** Sort the jobs in the following way: first the jobs of  $A$  in non-decreasing order of  $a_j$ , then the jobs of  $B$  in non-increasing order of  $b_j$ . Schedule the jobs on the two machines according to this order.

**Example:**

$A = \{\text{pie, cake}\}$

$B = \{\text{pizza}\}$

	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  $M_2$ .
  - $C_k = a_1 + a_2 + \dots + a_k + b_k$ .
  - $M_2$  is not idle when the rest of the jobs are processed, thus,  $C_{\max} = a_1 + \dots + a_k + b_k + b_{(k+1)} + \dots + b_n$ .
- 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  $F_2||C_{\max}$ .

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

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

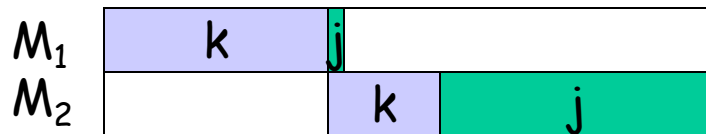
**Step:** Assume that Johnson rule is optimal for  $n-1$  jobs, and consider an instance with  $n$  jobs.

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

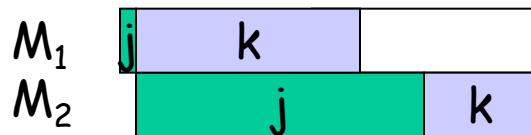
# Optimality of Johnson Rule for $F2||C_{\max}$

**Lemma:** If  $a_j=0$ , then there exist an optimal schedule in which  $j$  is first, If  $b_j=0$ , then there exists an optimal schedule in which  $j$  is last.

**Proof:** If  $a_j=0$  and  $J_j$  is not first, assume it is processed after  $J_k$ . The jobs  $J_j$  and  $J_k$  can be swapped without hurting the makespan.



The proof for  $b_j=0$  is similar



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

Back to induction step: Recall that  $J_j$  is a job with  $a_j=0$  or  $b_j=0$ .

If  $a_j=0$ , then there exist an optimal schedule in which  $j$  is first (and can be processed by  $M_2$  with no delay), and if  $b_j=0$ , 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.

