Deterministic Scheduling

Dr inż. Krzysztof Giaro
Gdańsk University of Technology

Lecture Plan

- Introduction to deterministic scheduling
- Critical path metod
- Some discrete optimization problems
- Scheduling to minimize $C_{\max}$
- Scheduling to minimize ΣC_i
- Scheduling to minimize $L_{\max}$
- Scheduling to minimize number of tardy tasks
- Scheduling on dedicated processors

Introduction to Deterministic Scheduling

Our aim is to ***schedule*** the given set of tasks (programs etc.) on ***machines*** (or processors).

We have to construct a schedule that fulfils given constraints and minimizes ***optimality criterion*** (objective function).

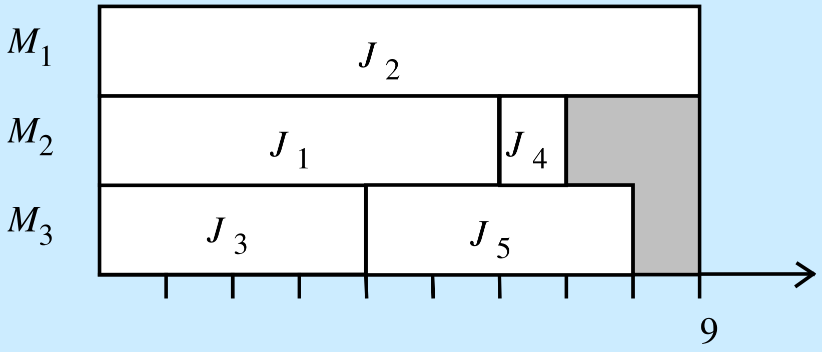
Deterministic model: all the parameters of the system and of the tasks are known in advance.

Genesis and practical motivations:

- scheduling manufacturing processes,
- project planning,
- school or conference timetabling,
- scheduling tasks in multitask operating systems,
- distributed computing.

Introduction to Deterministic Scheduling

Example 1. Five tasks with processing times $p_1,...,p_5$=6,9,4,1,4 have to be scheduled on three processors to minimize schedule length.



Graphical representation of a schedule - Gantt chart

Why the above schedule is feasible?

General **constriants in classical scheduling theory**:

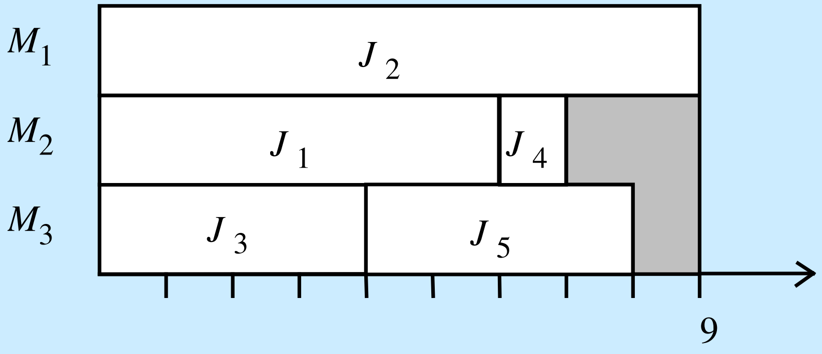
- each task is processed by at most one processor at a time,
- each processor is capable of processing at most one task at a time,
- other constraints - to be discussed ...

Introduction to Deterministic Scheduling

Processors characterization

Parallel processors (each processor is capable to process each task):

- ***identical processors*** – every processor is of the same speed,
- ***uniform processors*** – processors differ in speed, but the speed does not depend on the task,
- ***unrelated processors*** – the speed of the processor depend on the particular task processed.



Schedule on three parallel processors

Introduction to Deterministic Scheduling

Processors characterization

Dedicated processors

- Each job consists of the set of tasks preassigned to processors (job J_j consists of tasks T_{ij} preassigned to M_i, of processing time p_{ij}). The job is completed at the time the latest task is completed,
- some jobs may not need all the processors (***empty operations***),
- no two tasks of the same job can be scheduled in parallel,
- a processor is capable to process at most one task at a time.

There are three models of scheduling on dedicated processors:

- ***flow shop*** – all jobs have the same processing order through the machines coincident with machine numbering,
- ***open shop*** – the sequenice of tasks within each job is arbitrary,
- ***job shop*** – the machine sequence of each job is given and may differ between jobs.

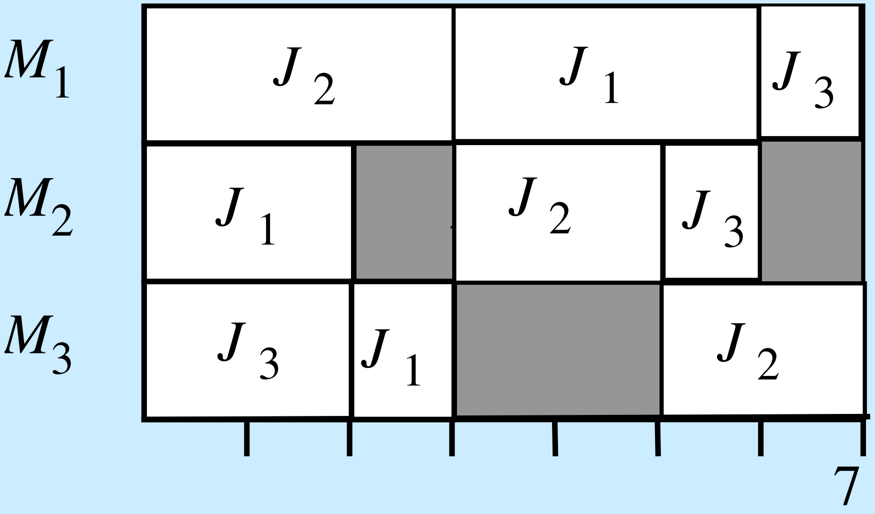
Introduction to Deterministic Scheduling

Processors characterization

Dedicated processors - open shop
(processor sequence is arbitrary for each job).

Example. One day school timetable.

		Teachers		
		M_1	M_2	M_3
Classes	J_1	3	2	1
	J_2	3	2	2
	J_3	1	1	2

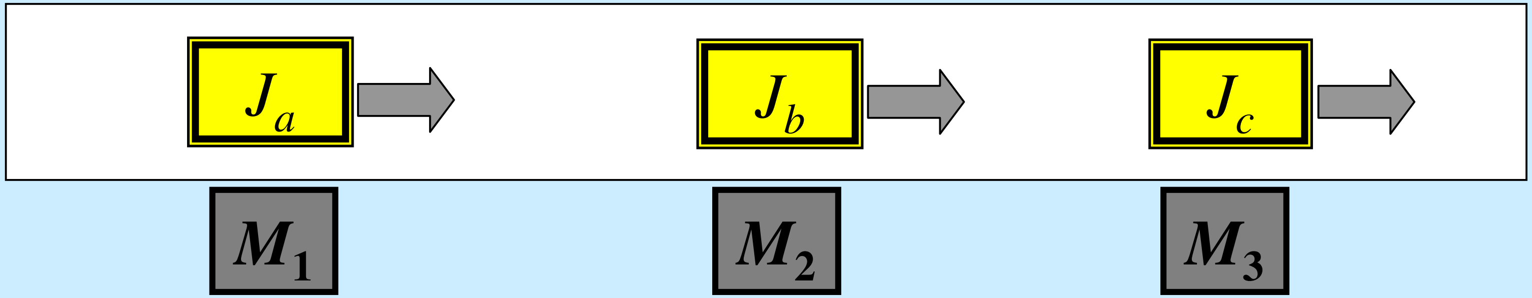


Introduction to Deterministic Scheduling

Processors characterization

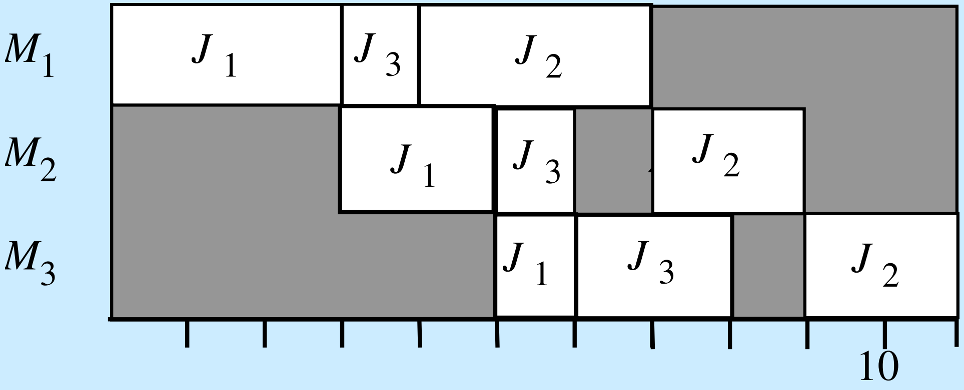
Dedicated processors - flow shop (processor sequence is the same for each job - task T_{ij} must precede T_{kj} for $i<k$).

Example. Conveyor belt.



Details

	M_1	M_2	M_3
J_1	3	2	1
J_2	3	2	2
J_3	1	1	2

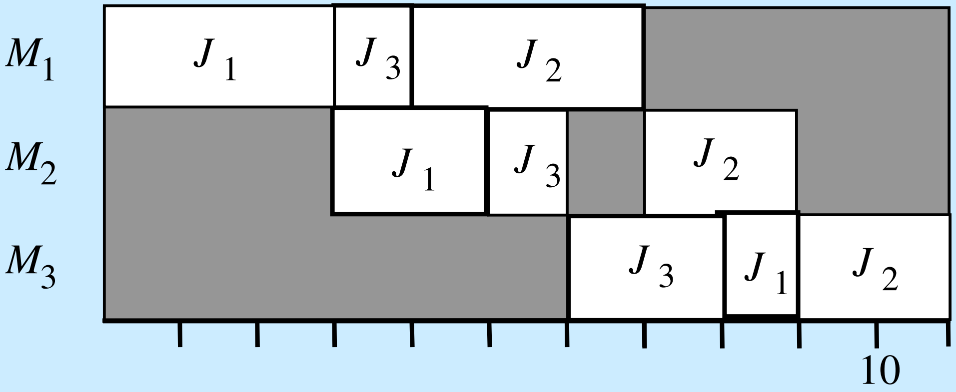


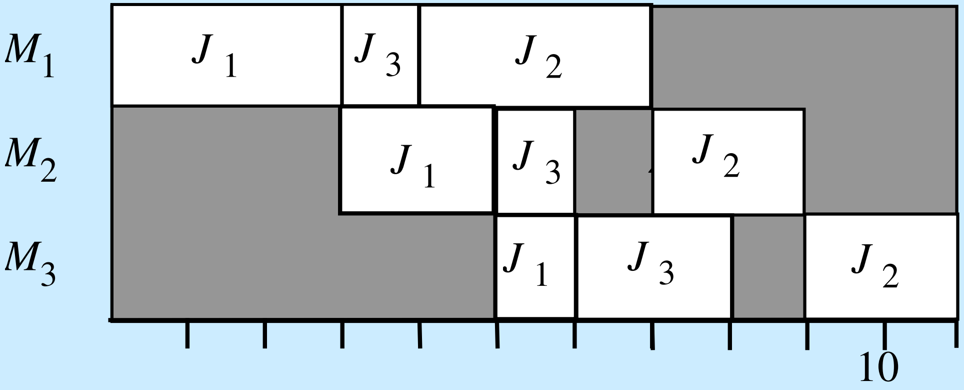
Introduction to Deterministic Scheduling

Processors characterization

Dedicated processors - flow shop (processor sequence is the same for each job - task T_{ij} must precede T_{kj} for $i<k$).

Flow shop allow the job order to differ between machines...





Permutation flow shop does not.

Dedicated processors will be considered later ...

Introduction to Deterministic Scheduling

Tasks characterization

There are given: the set of *n* tasks $T=\{T_1,\ldots,T_n\}$ and *m* machines (processors) $M=\{M_1,\ldots,M_m\}$.

Task T_j :

- ***Processing time***. It is independent of processor in case of identical processors and is denoted p_j. In the case of uniform processors, processor M_i speed is denoted b_i, and the processing time of T_j on M_i is p_j/b_i. In the case of unrelated processors the processing time of T_j on M_i is p_{ij}.
- ***Release (or arrival) time*** r_j. The time at which the task is ready for processing. By default all release times are zero.
- ***Due date*** d_j. Specifies the time limit by which should be completed. Usually, penalty functions are defined in accordance with due dates, or d_j denotes the ‘hard’ time limit (deadline) by which T_j **must** be completed (exact meaning comes from the context).
- ***Weight*** w_j – expresses the relative urgency of T_j, by default $w_j=1$.

Introduction to Deterministic Scheduling

Tasks characterization

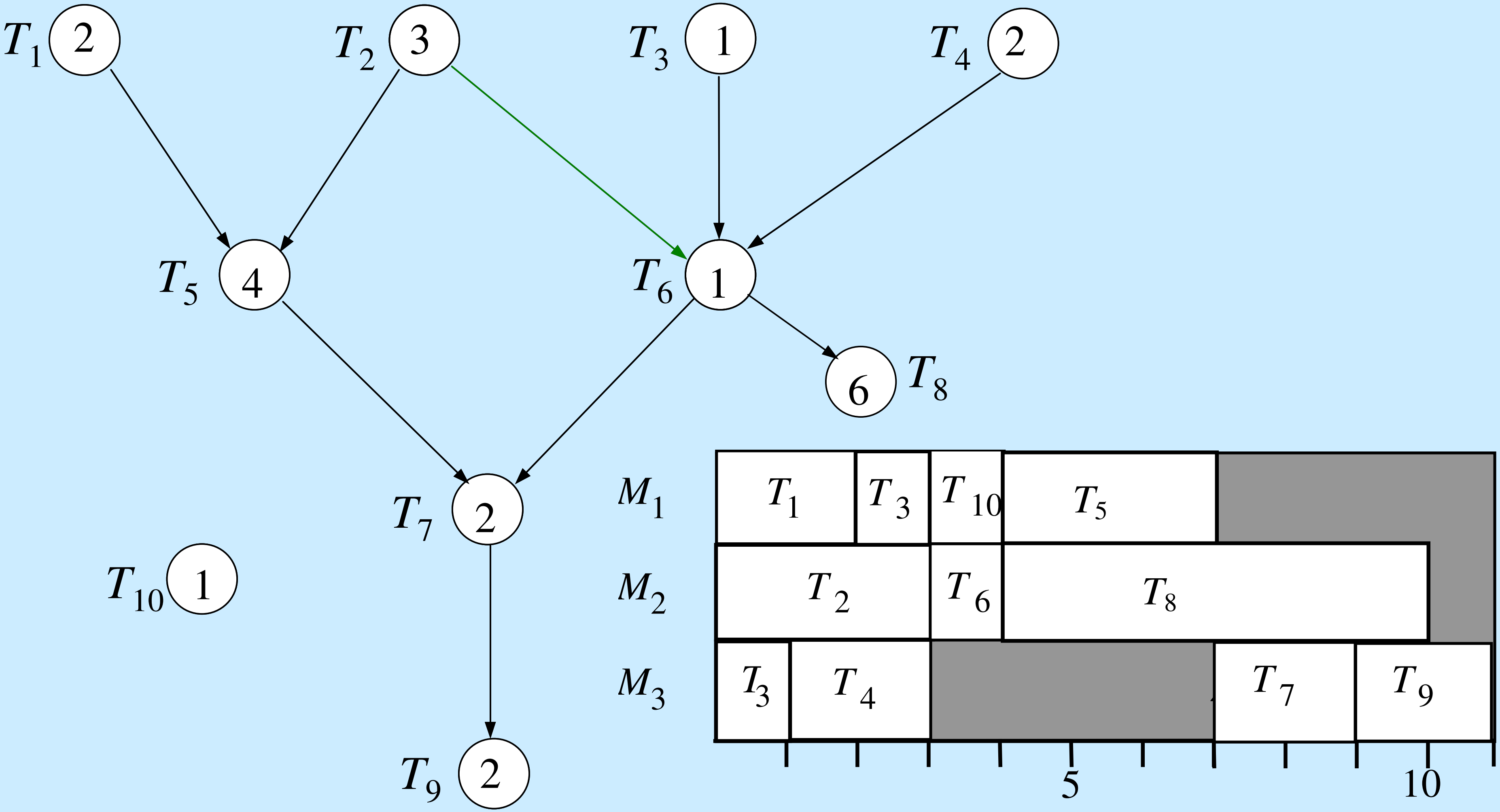
Dependent tasks:

- In the task set there are some ***precedence constraints*** defined by a precedence relation. $T_i \prec T_j$ means that task T_j cannot be started until T_i is completed (e.g. T_j needs the results of T_i).
- In the case there are no precedence constraints, we say that the tasks are ***independent*** (by default). In the other case we say the tasks are ***dependent***.
- The precedence relation is usually represented as a directed graph in shich nodes correspond to tasks and arcs represent precedence constraints (*task-on-node graph*). Transitive arcs are usually removed from precedence graph.

Introduction to Deterministic Scheduling

Tasks characterization

Example. A schedule for 10 dependent tasks (p_j given in the nodes).

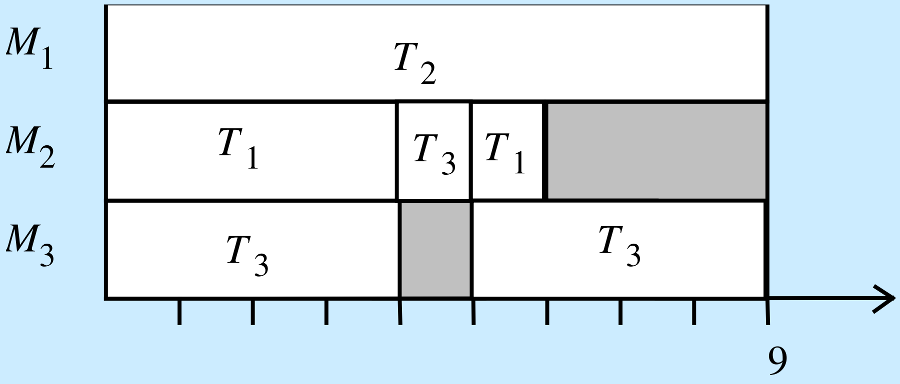


Introduction to Deterministic Scheduling

Tasks characterization

A schedule can be:

- ***non-preemptive*** – preempting of any task is not allowed (default),
- ***preemptive*** – each task may be preempted at any time and restarted later (even on a different processor) with no cost.



Preemptive schedule on parallel processors

Introduction to Deterministic Scheduling

Feasible schedule conditions (gathered):

- each processor is assigned to at most one task at a time,
- each task is processed by at most one machine at a time,
- task T_j is processed completly in the time interval $[r_j, \infty)$ (or within $[r_j, d_j)$, when deadlines are present),
- precedence constraints are satisfied,
- in the case of non-preemptive scheduling no task is preempted, otherwise the number of preemptions is finite.

Introduction to Deterministic Scheduling

Optimization criteria

A location of the task T_i within the schedule:

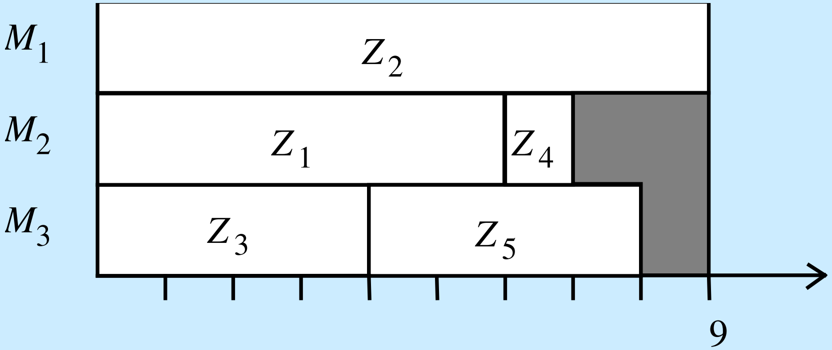
- ***completion time*** C_i,
- ***flow time*** $F_i = C_i – r_i$,
- ***lateness*** $L_i = C_i – d_i$,
- ***tardiness*** $T_i = \max\{C_i – d_i, 0\}$,
- "***tardiness flag***" $U_i = \mathrm{w}(C_i > d_i)$, i.e. the answer (0/1 logical yes/no) to the question whether the task is late or not.

Introduction to Deterministic Scheduling

Optimization criteria

Most common optimization criteria:

- ***schedule length (makespan)*** $C_{\max} = \max\{C_j : j=1,\ldots,n\}$,
- ***sum of completion times (total completion time)*** $\Sigma C_j = \Sigma_{i=1,\ldots,n} C_i$,
- ***mean flow time*** $\overline{F} = (\Sigma_{i=1,\ldots,n} F_i)/n$.



$$\mathbf{C_{\max} = 9,}$$

$$\mathbf{\Sigma C_j = 6+9+4+7+8 = 34}$$

A schedule of three parallel processors. $p_1,\ldots,p_5$=6,9,4,1,4.

In the case there are tasks weights we can consider:

- *sum of weighted completion times* $\Sigma w_j C_j = \Sigma_{i=1,\ldots,n} w_i C_i$,

$w_1,\ldots,w_5$=1,2,3,1,1 $\mathbf{\Sigma w_j C_j = 6+18+12+7+8 = 51}$

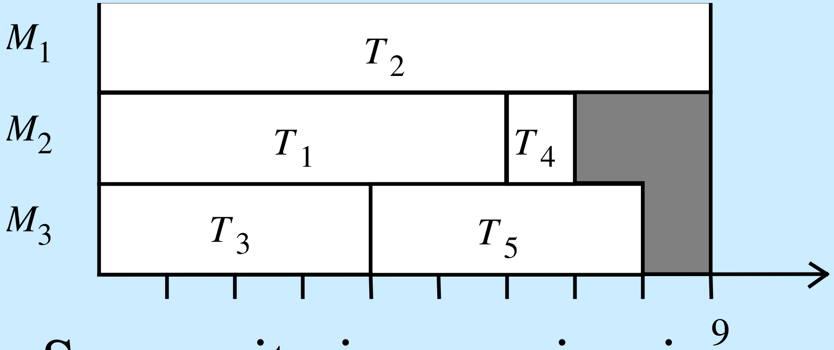
Introduction to Deterministic Scheduling

Optimization criteria

Related to due times:

- ***maximum lateness*** $L_{\max} = \max\{L_j : j=1,\ldots,n\}$,
- ***maximum tardiness*** $T_{\max} = \max\{T_j : j=1,\ldots,n\}$,
- ***total tardiness*** $\Sigma T_j = \Sigma_{i=1,\ldots,n} T_i$,
- ***number of tardy tasks*** $\Sigma U_j = \Sigma_{i=1,\ldots,n} U_i$,
- weighted criteria may be considered, e.g. *total weighted tardiness*

$\Sigma w_j T_j = \Sigma_{i=1,\ldots,n} w_i T_i.$



Task:	T_1	T_2	T_3	T_4	T_5
$d_i=$	7	7	5	5	8
$L_i=$	-1	2	-1	2	0
$T_i=$	0	2	0	2	0

$\mathbf{L_{\max} = T_{\max} = 2}$

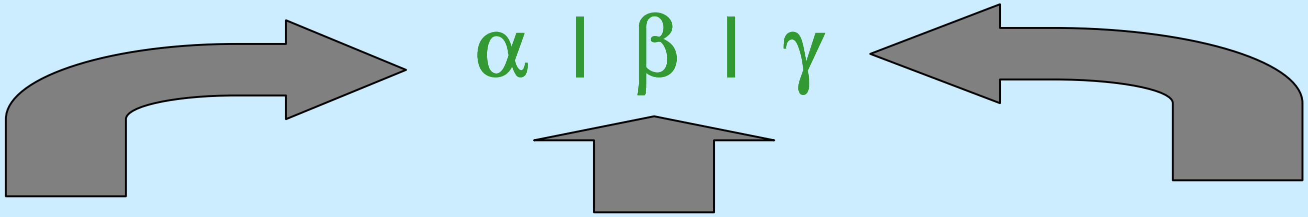
$\mathbf{\Sigma T_j = 4, \Sigma U_j = 2}$

Some criteria are pair-wise equivalent

$\Sigma L_i = \Sigma C_i - \Sigma d_i$, $\overline{F} = (\Sigma C_i)/n - (\Sigma r_i)/n.$

Introduction to Deterministic Scheduling

Classification of deterministic scheduling problems.



$\alpha \mid \beta \mid \gamma$

Procession environment

Task characteristics

Optimization criterion

α is of the form:

- *P* – identical processors
- *Q* – uniform processors
- *R* – unrelated processors
- *O* – open shop
- *F* – flow shop
- *PF* – „permutation” flow shop
- *J* – job shop

Moreover:

- there may be specified the number of processors e.g. $O4$,
- in the case of single processors we just put 1,
- we put ‘–’ in the case of processor-free environment.

Introduction to Deterministic Scheduling

Classification of deterministic scheduling problems.

In the case β is empty all tasks characteristics are default: the tasks are non-preemptive, not dependent r_j=0, processing times and due dates are arbitrary.

β possible values:

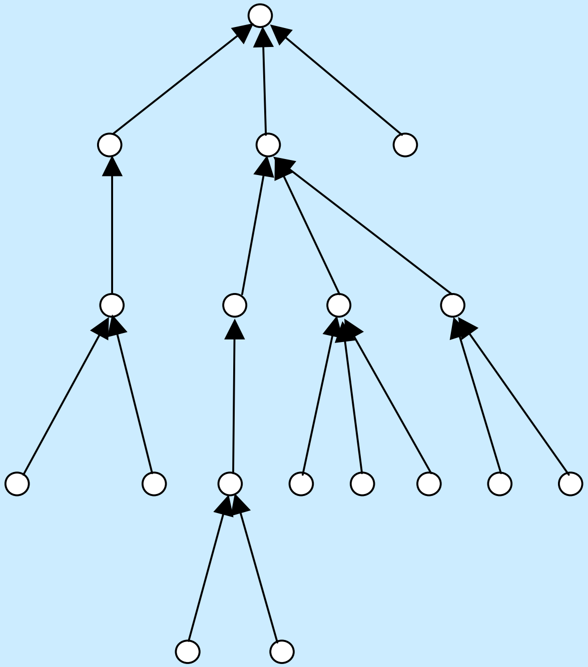
- *pmtn* – preemptive tasks,
- *res* – additional resources required (omitted),
- *prec* – there are precedence constraints,
- r_j – arrival times differ per task,
- p_j=1 or *UET* – all processing times equal to 1 unit,
- $p_{ij}\in\{0,1\}$ or *ZUET* – all tasks are of unit time or empty (dedicated processors),
- $C_j \leq d_j$ or d_j denote deadlines,

Introduction to Deterministic Scheduling

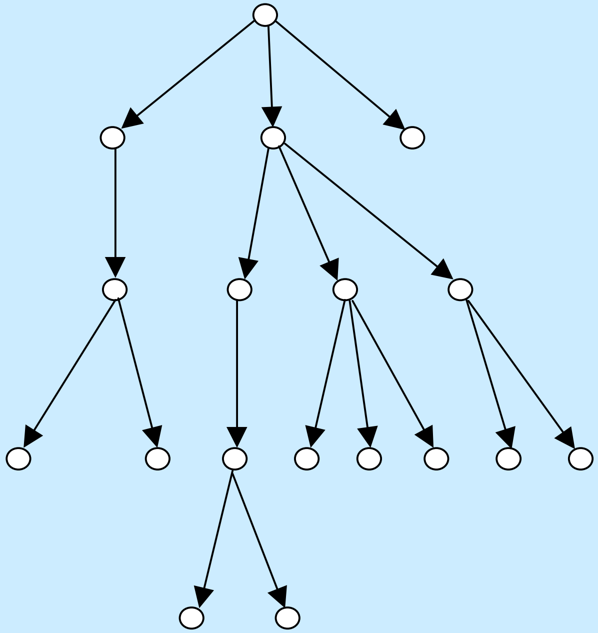
Classification of deterministic scheduling problems.

β possible values:

- *in–tree, out–tree, chains* ... – reflects the precedence constraints (*prec*).



in–tree



out–tree

Introduction to Deterministic Scheduling

Classification of deterministic scheduling problems.

Examples.

$P3|prec|C_{\max}$ – scheduling non-preemptive tasks with precedence constraints on three parallel identical processors to minimize schedule length.

$R|pmtn,prec,r_i|\Sigma U_i$ – scheduling preemptive dependent tasks with arbitrary ready times and arbitrary due dates on parallel unrelated processors to minimize the number ot tardy tasks.

$1|r_i,C_i \le d_i|–$ – decision problem of existence (no optimization criterion) of schedule of independent tasks with arbitrary ready times and deadlines on a single processor, such that no task is tardy.

Introduction to Deterministic Scheduling

Propertiels of computer algorithm evaluating its quality.

Computational (time) complexity – function that estimates (upper bound) the worst-case operation number performed during execution in terms of input data size.

Polynomial (time) algorithm– when time complexity may be bounded by some polynomial of data size. In computing theory polynomial algorithms are considered as efficient.

Introduction to Deterministic Scheduling

Propertiels of computer algorithm evaluating its quality.

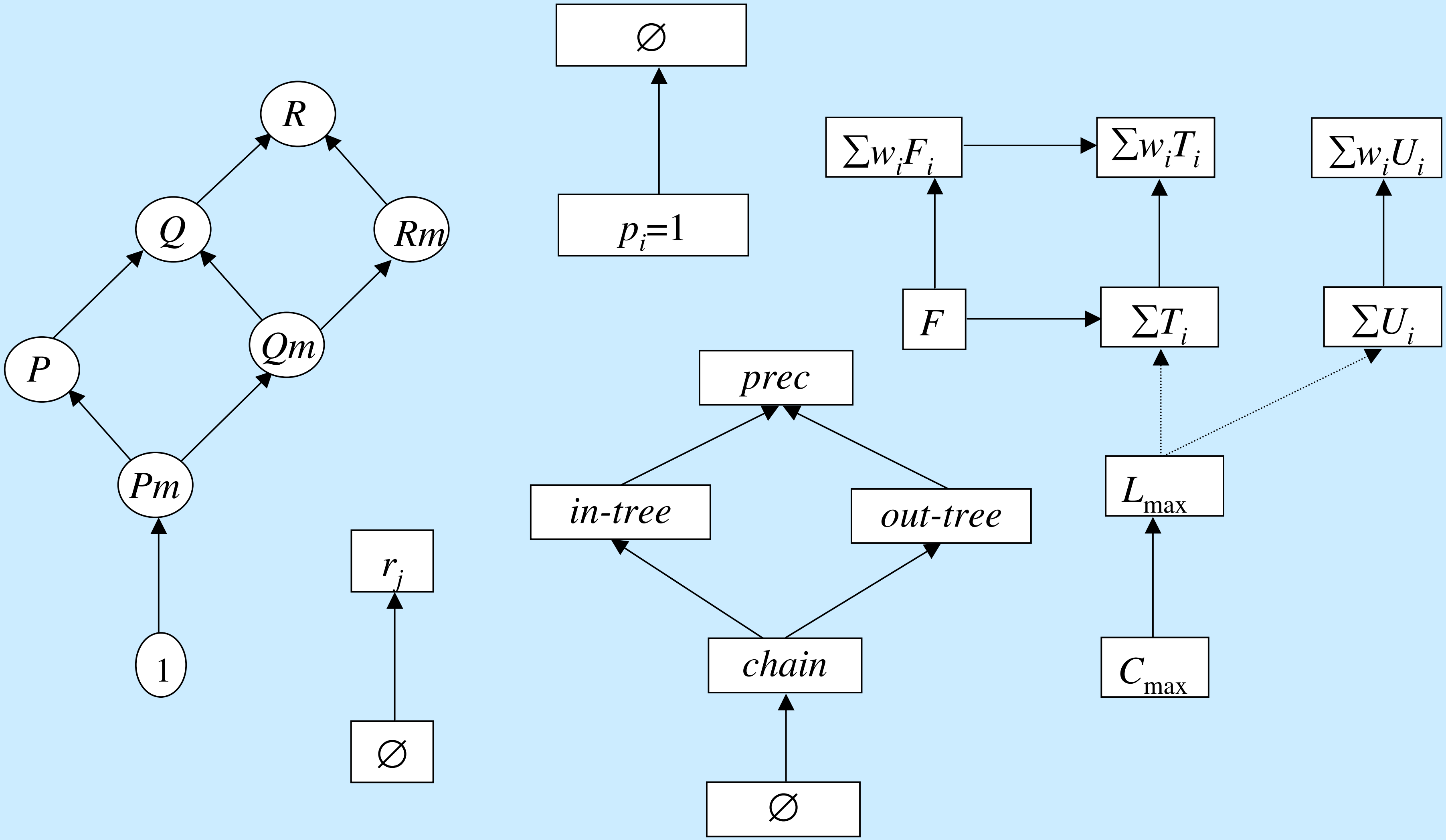
NP-hard problems – are commonly believed not to have polynomial time algorithms solving them. For these problems we can only use fast but not accurate procedures or (for small instances) long time heuristics. NP-hardness is treated as computational intractability.

How to prove that problem A is NP-hard?

Sketch: Find any NP-hard problem B and show the efficient (polynomial) procedure that reduces (translates) B into A. Then A is not less general problem than B, therefore if B was hard, so is A.

Introduction to Deterministic Scheduling

Reductions of scheduling problems



R

Q

Rm

P

Qm

Pm

1

r_j

$\varnothing$

$\varnothing$

$p_i=1$

$prec$

$in\text{-}tree$

$out\text{-}tree$

$chain$

$\varnothing$

$\sum w_i F_i$

$\sum w_i T_i$

$\sum w_i U_i$

F

$\sum T_i$

$\sum U_i$

$L_{\max}$

$C_{\max}$

Introduction to Deterministic Scheduling

Computational complexity of scheduling problems

In we restrict the number of processors to 1,2,3,•, there are 4536 problems:

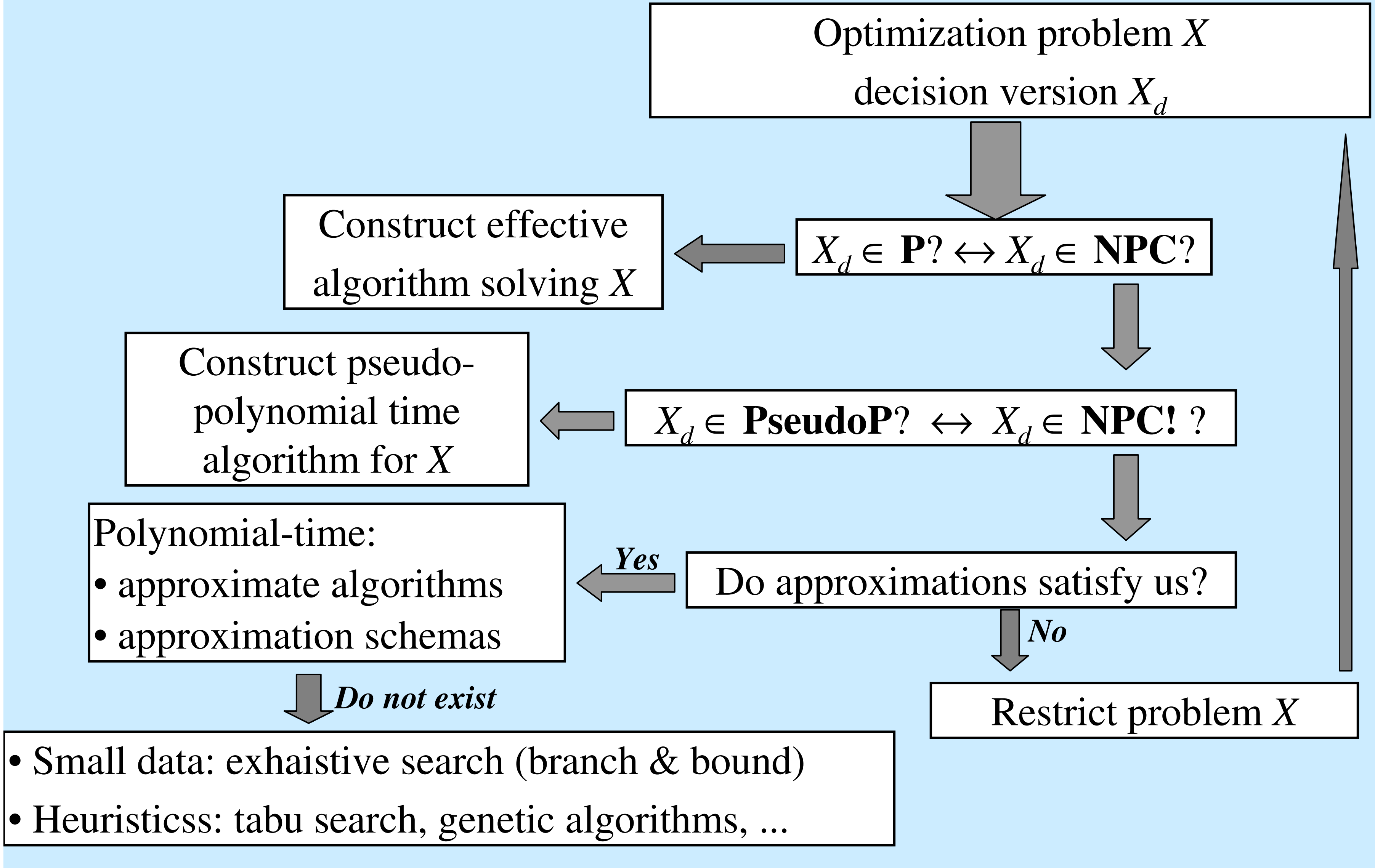
- 416 – polynomial-time solvable,
- 3817 – NP–hard,
- 303 – open.

How do we cope with NP-hardness?

- exact *pseudo-polynomial time algorithms*,
- exact algorithms, efficient only in the *mean-case*,
- *heuristics* (*tabu-search, genetic algorithms etc*.),
- in the case of small input data - exponential *exhaustive search* (e.g. *branch–bound*).

Introduction to Deterministic Scheduling

General problem analysis schema



Optimization problem X

decision version X_d

$X_d \in$ **P**? $\leftrightarrow$ $X_d \in$ **NPC**?

Construct effective algorithm solving X

$X_d \in$ **PseudoP**? $\leftrightarrow$ $X_d \in$ **NPC!** ?

Construct pseudo-polynomial time algorithm for X

Do approximations satisfy us?

Yes

Polynomial-time:
- approximate algorithms
- approximation schemas

Do not exist

- Small data: exhaistive search (branch & bound)
- Heuristicss: tabu search, genetic algorithms, ...

No

Restrict problem X

Critical path method.

$-|prec|C_{\max}$ model consists of a set of dependent tasks of arbitrary lengths, which do not need processors. Our aim is to construct a schedule of minimum length.

Precedence relation $\prec$ is a **quasi order** in the set of tasks, i.e. it is:

- anti-reflective: $\forall_{T_i} \neg T_i \prec T_i$
- transistive: $\forall_{T_i, T_j, T_k} \left(T_i \prec T_j \wedge T_j \prec T_k\right) \Rightarrow T_i \prec T_k$

Critical path method.

Precence relation $\prec$ is represented with an ***acyclic digraph***.

AN *(activity on node)* network:

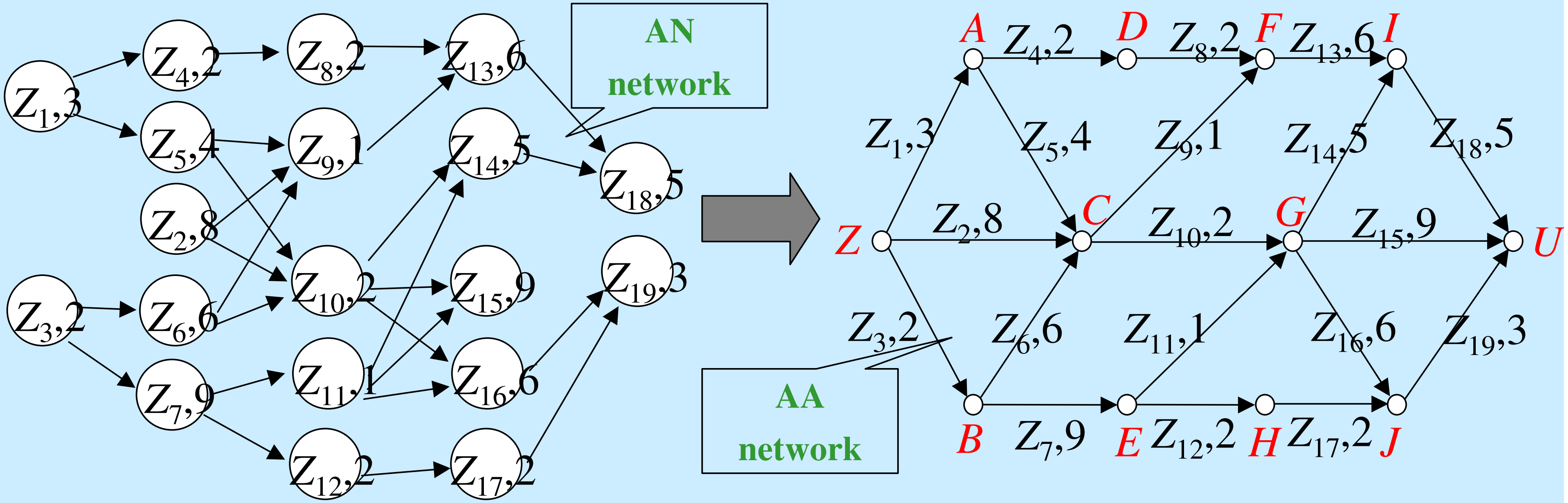
- nodes correspond to tasks, nodes weights equal to processing times,
- $T_i \prec T_j \Leftrightarrow$ there exists a directed path connecting node T_i and node T_j,
- transitive arcs are removed.

AA *(activity on arc)* network:

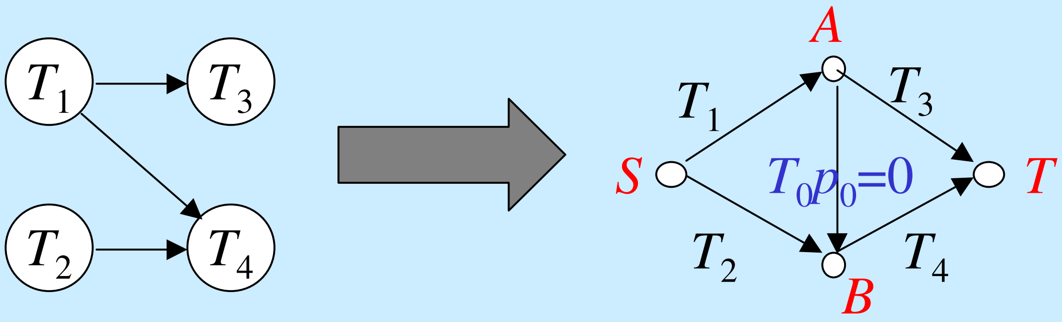
- arcs correspond to tasks, their length is equal to processing times,
- for each node v there exists a path starting at S (source) and terminating at T (sink) passing through v,
- $T_i \prec T_j \Leftrightarrow$ arc T_i end-node is the starting-node of T_j, or there exists a directed path starting at T_i end-node and terminating at T_j start-node.,
- to construct the network one may need to add *apparent tasks* – zero-length tasks.

Critical path method.

Example. Precedence relation for 19 tasks.



Example. Translating AN to AA we may need to add (zero-length) apparent tasks.



Critical path method.

–|*prec*|$C_{\max}$ model consists of a set of dependent tasks of arbitrary lengths, which do not need processors. Our aim is to construct a schedule of minimum length.

The idea: for every task T_i we find the earliest possible start time $l(T_i)$, i.e. the length of the longest path terminating at that task.

How to find these start times?

AN network Algorithm:

1. find a topological node ordering (the start of any arc precedes its end),
2. assign $l(T_a)$=0 for every task T_a without predecessor,
3. assing $l(T_a)=\max\{l(T_j)+p_j$: exists an arc$(T_j, T_i)\}$ to all other tasks in topological order.

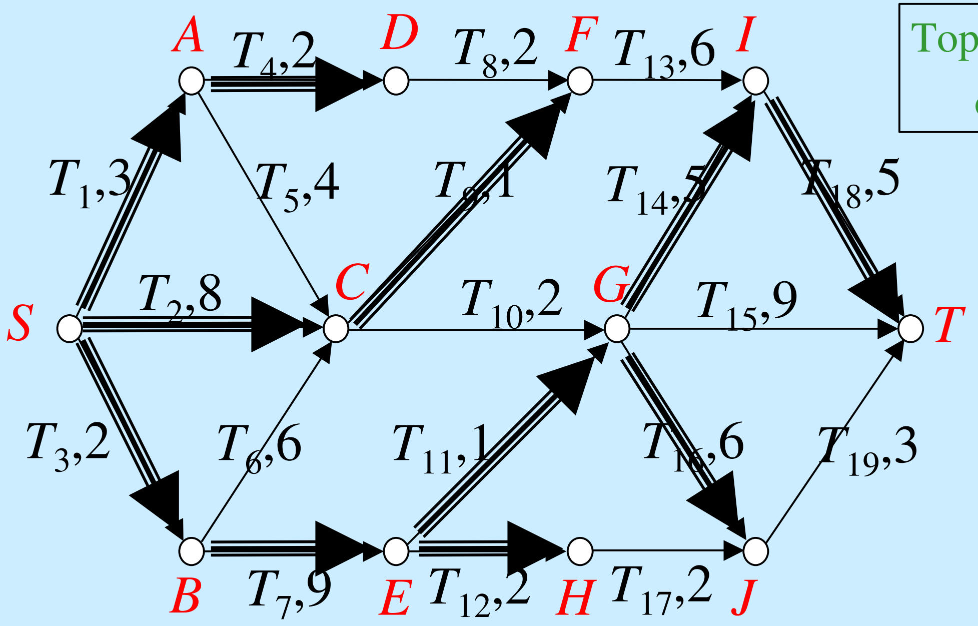
AA network Algorithm:

1. find a topological node ordering,
2. $l(S)=0$, assign $l(v)=\max\{l(u)+p_j$: arc T_j connects u and $v\}$ to each node v,

Result: $l(T_j)$ is equal to $l(v)$ of the starting node v of T_j. $l(T)$ is the length of an optimal schedule.

Critical path method.

Example: construction of a schedule for 19 tasks.

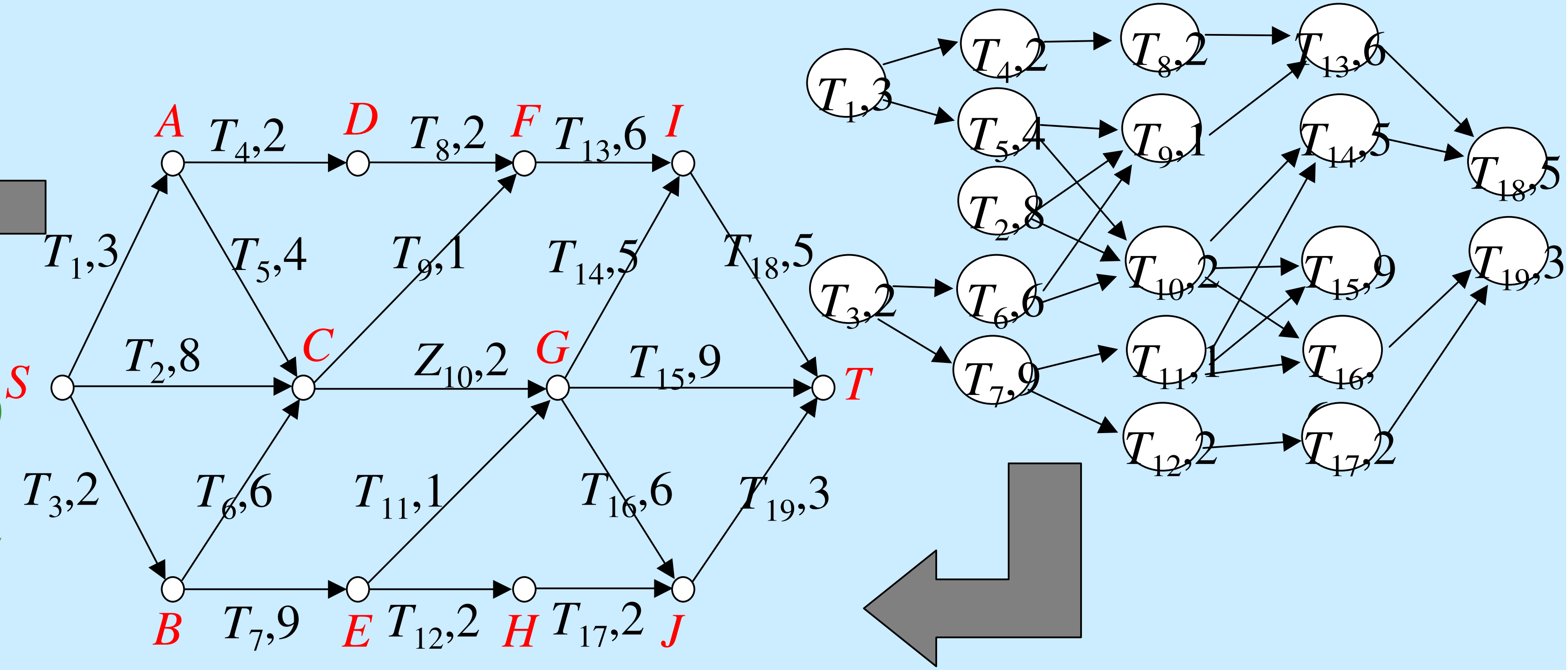


Topological order

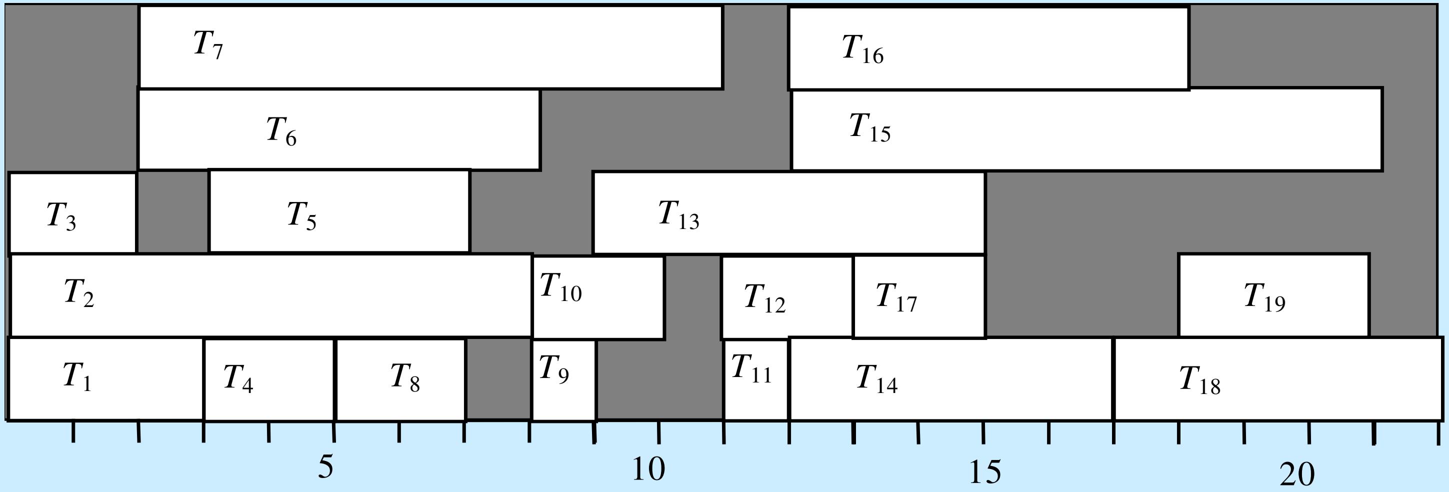
Starting times

S:	0	
A:	3	Z:0+3
B:	2	Z:0+2
C:	8	Z:0+8, A:3+4, B:2+6
D:	5	A:3+2
E:	11	B:2+9
F:	9	C:8+1 ,D:5+2
G:	12	C:8+2, E:11+1
H:	13	E:11+2
I:	17	F:9+6, G:12+5
J:	18	G:12+6, H:13+2
T:	22	I:17+5, J:18+3, G:12+9

Critical path method.



S:	0
A:	3
B:	2
C:	8
D:	5
E:	11
F:	9
G:	12
H:	13
I:	17
J:	18
T:	22



Critical path method.

- Critical path method does not only minimize $C_{\max}$, but also optimizes all previously defined criteria.
- We can introduce to the model arbitrary release times by adding for each task T_j extra task of length r_j preceding T_j.

Some Discrete Optimization Problems

- ***maximum flow problem***. There is given a loop-free multidigraph $D(V,E)$ where each arc is assigned a capacity $w{:}E\rightarrow N$. There are two specified nodes - the source s and the sink t. The aim is to find a ***flow*** $f{:}E\rightarrow N\cup\{0\}$ of maximum ***value***.

What is a flow of value F?

- $\forall_{e\in E}\, f(e)\leq c(e)$, (flows may not exceed capacities)
- $\forall_{v\in V-\{z,u\}}\, \Sigma_{e \text{ terminates at } v}\, f(e) - \Sigma_{e \text{ starts at } v}\, f(e) = 0$,

(the same flows in and flows out for every ‘ordinary’ node)

- $\Sigma_{e \text{ terminates at } t}\, f(e) - \Sigma_{e \text{ starts at } t}\, f(e) = F$,

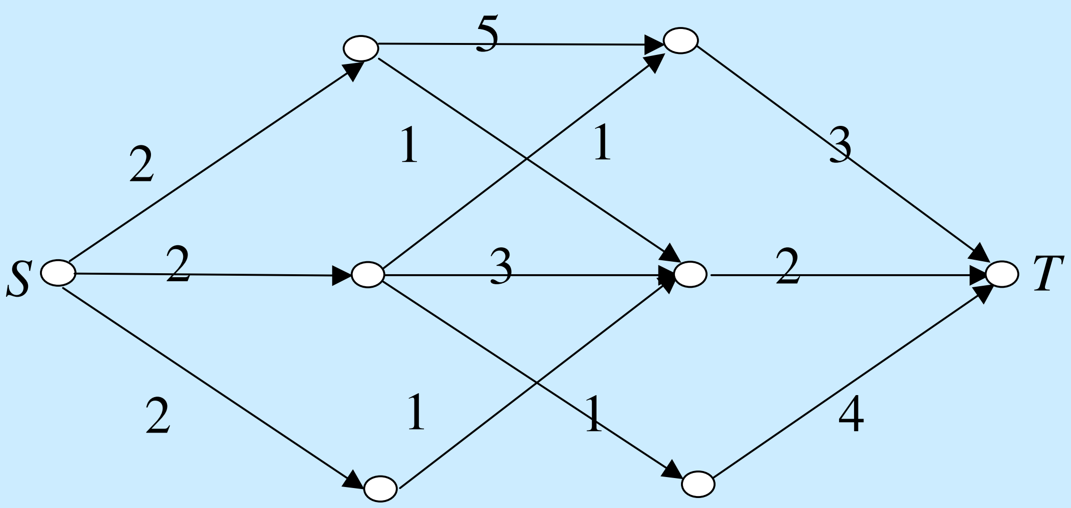
(F units flow out of the network through the sink)

- $\Sigma_{e \text{ terminates at } s}\, f(e) - \Sigma_{e \text{ starts at } s}\, f(e) = -F$.

(F units flow into the network through the source)

Some Discrete Optimization Problems

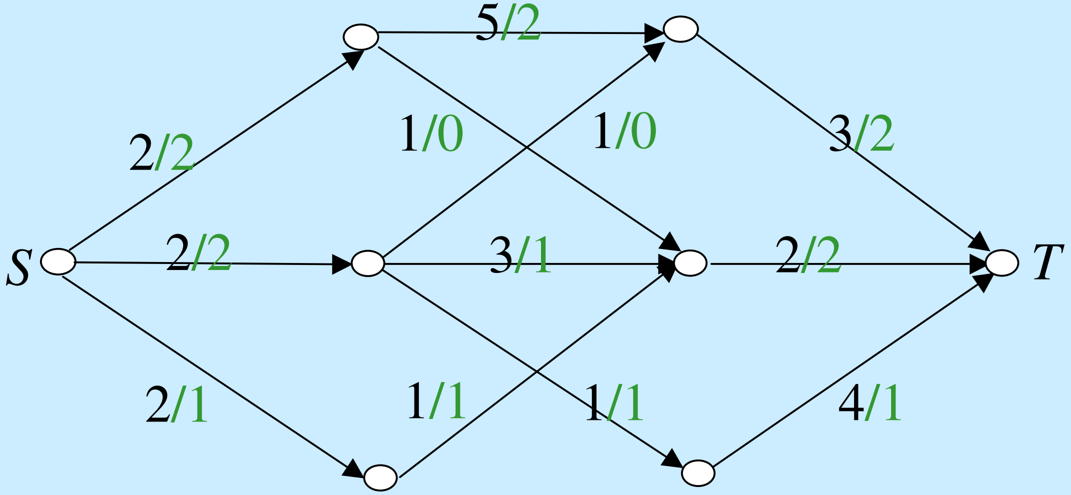
- ***maximum flow problem***. There is given a loop-free multidigraph $D(V,E)$ where each arc is assigned a capacity $w:E\rightarrow N$. There are two specified nodes - the source s and the sink t. The aim is to find a ***flow*** $f:E\rightarrow N\cup\{0\}$ of maximum ***value***.



Network, arcs capacity

Some Discrete Optimization Problems

- ***maximum flow problem***. There is given a loop-free multidigraph $D(V,E)$ where each arc is assigned a capacity $w:E\rightarrow N$. There are two specified nodes - the source s and the sink t. The aim is to find a ***flow*** $f:E\rightarrow N\cup\{0\}$ of maximum ***value***.



... and maximum flow $F=5$

Complexity $O(|V||E|\log(|V|^2|E|) \leq O(|V|^3)$.

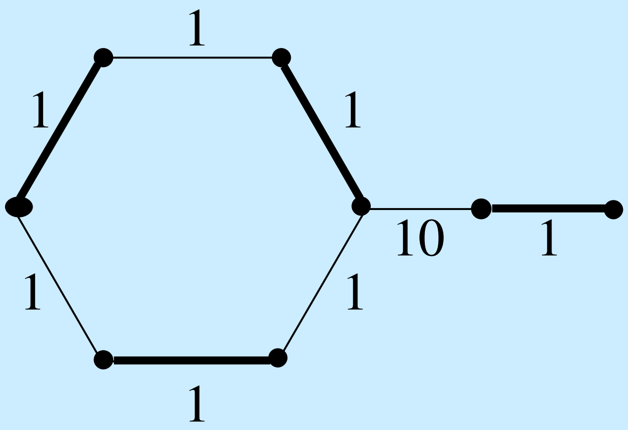
Some Discrete Optimization Problems

- Many ***graph coloring*** models.
- ***Longest (shortest) path*** problems.
- ***Linear programming*** – polynomial-time algorithm known.

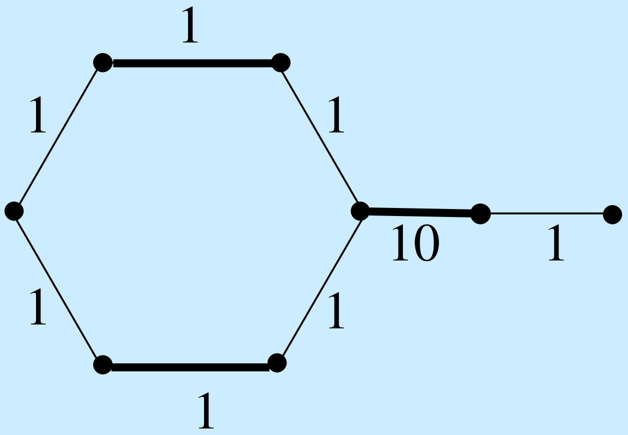
- The problem of ***graph matching***. There is given graph $G(V,E)$ with a weight function $w:E\rightarrow N\cup\{0\}$. A ***matching*** is a subset $A\subset E$ of pair-wise non-neighbouring edges.

- ***Maximum matching***: find a matching of the maximum possible cardinality ($\alpha(L(G))$). **The complexity $O(|E||V|^{1/2})$.**
- ***Heaviest (lightest) matching of a given cardinality***. For a given $k\leq\alpha(L(G))$ find a matching of cardinality k and maximum (minmum) possible weight sum.
- ***Heaviest matching***. Find a matching of maximum possible weight sum. **The complexity $O(|V|^3)$** for bipartite graphs and **$O(|V|^4)$** in general case.

Some Discrete Optimization Problems



Cardinality: 4
Weight: 4



Cardinality: 3
Weight: 12

Maximum matching needs not to be the heaviest one and vice-versa.

Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, independent tasks

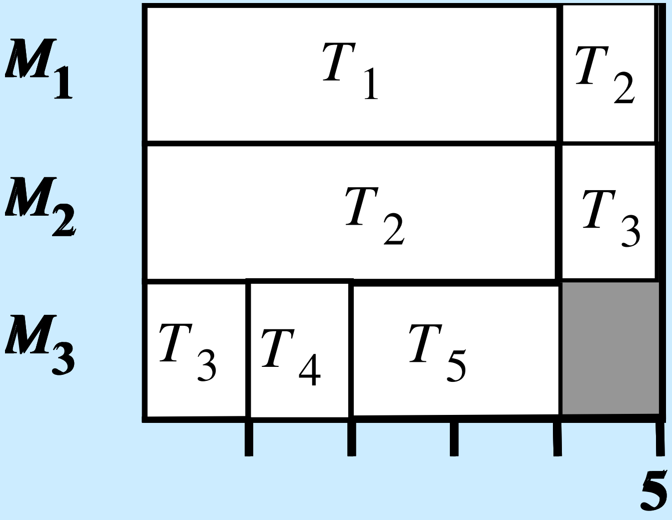
Preemptive scheduling **$P|pmtn|C_{\max}$.**

McNaughton Algorithm (complexity $\boldsymbol{O(n)}$)

1. Derive optimal length $C_{\max}^*=\max\{\Sigma_{j=1,\ldots,n}\, p_j/m,\ \max_{j=1,\ldots,n} p_j\}$,
2. Schedule the consecutive tasks on the first machine until $C_{\max}^*$ is reached. Then interrupt the processed task (if it is not completed) and continue processing it on the next machine starting at the moment 0.

Example. m=3, n=5, $p_1,\ldots,p_5$=4,5,2,1,2.

$\Sigma_{i=1,\ldots,5}\, p_i=14$, $\max p_i=5$,
$C_{\max}^*=\max\{14/3,5\}=5$.



Scheduling on Parallel Processors to Minimize the Schedule Length.

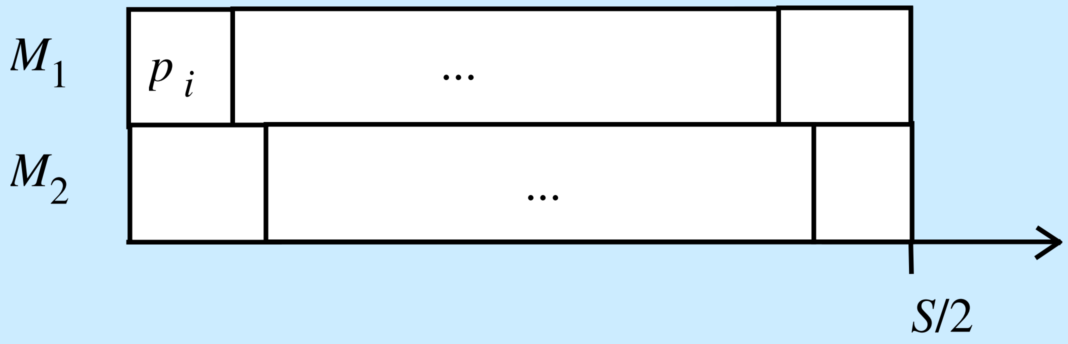
Identical processors, independent tasks

Non-preemptive scheduling $P||C_{\max}$.

The problem is NP-hard even in the case of two processors ($P2||C_{\max}$).

Proof. *Partition Problem*: there is given a sequence of positive integers $a_1,\ldots a_n$ such that $S=\Sigma_{i=1,\ldots,n}\, a_i$. Determine if there exists a sub-sequence of sum $S/2$?

$PP \Rightarrow P2||C_{\max}$ reduction: put n tasks of lengths $p_j=a_j$ ($j=1,\ldots,n$), and two processors. Determine if $C_{\max} \leq S/2$.



There exists an exact pseudo-polynomial dynamic programming algorithm of complexity $O(nC^m)$, for some $C \geq C_{\max}{}^*$.

Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, independent tasks

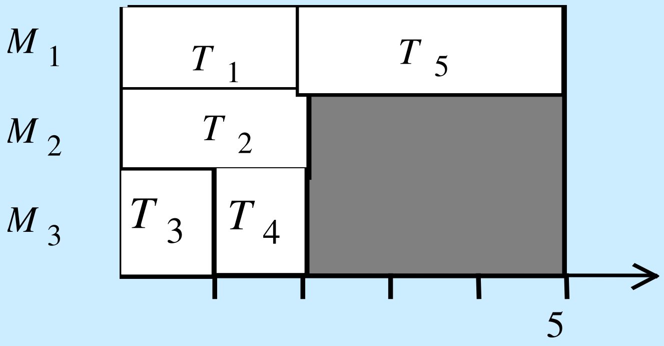
Non-preemptive scheduling $P||C_{\max}$.

Polynomial-time approximation algorithms.

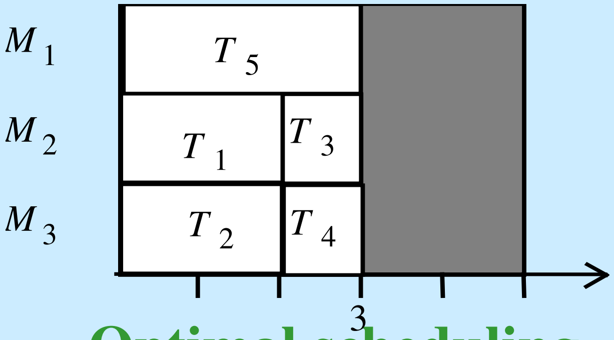
List Scheduling LS - an algorithm used in numerous problems:

- fix an ordering of the tasks on the list,
- any time a processor gets free (a task processed by that processor has been completed), schedule the first *available* task from the list on that processor.

Example. $m=3$, $n=5$, $p_1,\ldots,p_5=2,2,1,1,3$.



List scheduling



Optimal scheduling

Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, independent tasks

Non-preemptive scheduling $P||C_{\max}$.

Polynomial-time approximation algorithms.

List Scheduling LS - an algorithm used in numerous problems:

- fix an ordering of the tasks on the list,
- any time a processor gets free (a task processed by that processor has been completed), schedule the first *available* task from the list on that processor.

Approximation ratio. LS is 2–approximate: $C_{\max}(\mathrm{LS}) \le (2-m^{-1})C_{\max}^*$.

Proof (includes dependent tasks model $P|prec|C_{\max}$). Consider a sequence of tasks $T_{\pi(1)}$, ... $T_{\pi(k)}$ in a LS schedule, such that $T_{\pi(1)}$ - the last completed task, $T_{\pi(2)}$ - the last completed predecessor of $T_{\pi(1)}$ etc.

$$C_{\max}^{(pmtn)} \le C_{\max}^* \le C_{\max}(\mathrm{LS}) \le \sum_{i=1,\dots,k} p_{\pi(k)} + \sum_{i\notin\pi} p_i/\mathrm{m}$$
$$=(1-1/\mathrm{m}) \sum_{i=1,\dots,k} p_{\pi(k)} + \sum_i p_i/\mathrm{m} \le (2-m^{-1})\, C_{\max}^{(pmtn)} \le (2-m^{-1})\, C_{\max}^*$$

Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, independent tasks

Non-preemptive scheduling $P||C_{\max}$.

Polynomial-time approximation algorithms.

Approximation ratio. LS is 2–approximate: $C_{\max}(\mathrm{LS}) \leq (2-m^{-1}) C_{\max}^{*}$.

Proof

$C_{\max}^{(pmtn)} \leq C_{\max}^{*}$ (preemptions may help)

$C_{\max}^{*} \leq C_{\max}(\mathrm{LS})$ (optimal vs non-optimal)

$C_{\max}(\mathrm{LS}) \leq \sum_{i=1,\ldots,k} p_{\pi(k)} + \sum_{i \notin \pi} p_i / m$ ($p_{\pi(1)}$ is the last completed job)

$$\sum_{i=1,\ldots,k} p_{\pi(k)} + \sum_{i \notin \pi} p_i / m = (1-1/m) \sum_{i=1,\ldots,k} p_{\pi(k)} + \sum_i p_i / m$$

$$(1-1/m) \sum_{i=1,\ldots,k} p_{\pi(k)} + \sum_i p_i / m \leq (2-m^{-1})\, C_{\max}^{(pmtn)}$$

because $(1-m^{-1}) \sum_{i=1,\ldots,k} p_{\pi(k)} \leq (1-m^{-1})\, C_{\max}^{(pmtn)}$ (prec. constraints)

$(2-m^{-1})\, C_{\max}^{(pmtn)} \leq (2-m^{-1})\, C_{\max}^{*}$ (see first step)

Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, independent tasks

Non-preemptive scheuling ***$P||C_{\max}$***.

Polynomial-time approximation algorithms.

LPT (*Longest Processing Time*) **scheduling**:

- List scheduling, where the tasks are sorted in non-increasing processing times p_i order.

Approximation ratio.LS is 4/3–approximate:

$$C_{\max}(\mathrm{LPT})\leq(4/3-(3m)^{-1})C_{\max}^*.$$

Unrelated processors, not dependent tasks

Preemptive scheduling ***$R|pmtn|C_{\max}$***

Polynomial time algorithm - to be discussed later ...

Non-preemptive scheduling ***$R||C_{\max}$***

- The problem is NP–hard (generalization of $P||C_{\max}$).
- Subproblem $Q|p_i=1|C_{\max}$ is solvable in polynomial time.
- LPT is used in practice.

Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

Preemptive scheduling $P|pmtn,prec|C_{\max}$.

- The problem is NP–hard.
- $P2|pmtn,prec|C_{\max}$ i $P|pmtn,forest|C_{\max}$ are solvable in $O(n^2)$ time.
- The following inequality estimating preemptive, non-preemptive and LS schedule holds:

$$C^*_{\max} \le (2-m^{-1}) C^{*\,(pmtn)}_{\max}$$

Proof. The same as in the case of not dependent tasks.

Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

Non-preemptive scheduling $P|prec|C_{\max}$.

- Obviously the problem is NP-hard.
- Many unit-time processing time cases are known to be solvable in polynomial time:
  - $P|p_i=1,in\text{–}forest|C_{\max}$ and $P|p_i=1,out\text{–}forest|C_{\max}$ (***Hu algorithm***, complexity $O(n)$),
  - $P2|p_i=1,prec|C_{\max}$ (***Coffman-Graham algorithm***, complexity $O(n^2)$),
- Even $P|p_i=1,opositing\text{–}forest|C_{\max}$ and $P|p_i=\{1,2\},prec|C_{\max}$ is NP–hard.

Hu algorithm:

- *out–forest* ➔ *in–forest reduction*: reverse the precedence relation. Solve the problem and reverse the obtained schedule.
- *in–forest* ➔ *in–tree*: add extra task dependent of all the roots. After obtaining the solution, remove this task from the schedule.
- Hu algorithm sketch: list scheduling for dependent tasks + descending distance from the root order.

Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

Non-preemptive scheduling

Hu algorithm (**$P|p_i=1,in\text{–}tree|C_{\max}$**):

- *Level of a task* – number of nodes in the path to the root.
- The task is *avaliable at the moment t* if all the tasks dependent of that task have been completed until *t*.

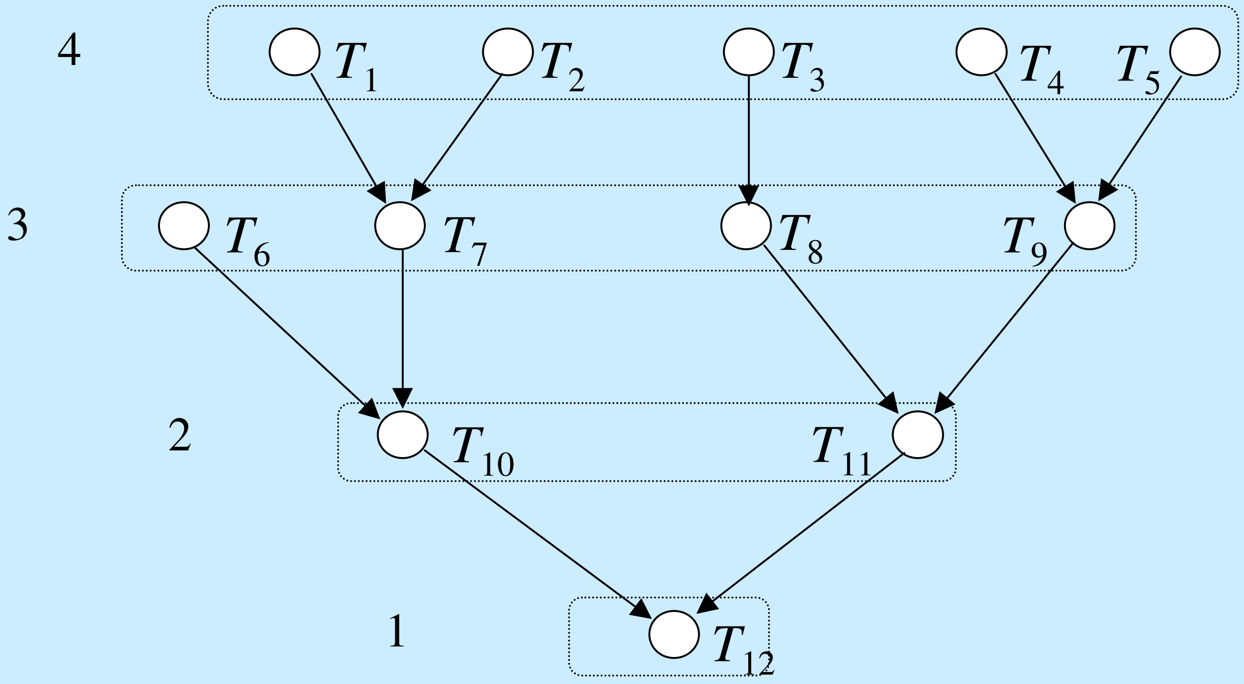
```
Compute the levels of the tasks;
$t$:=1;
repeat
        Find the list $L_t$ of all tasks avalable at the moment $t$;
        Sort $L_t$ in non-increasing levels order;
        Assign $m$ (or less) forst tasks from $L_t$ to the processors;
        Remove the scheduled tasks from the graph;
        $t$:=$t$+1;
until all the tasks are scheduled;
```

Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

<u>Non-preemptive scheduling</u>

Example. Hu algorithm. n=12, m=3.



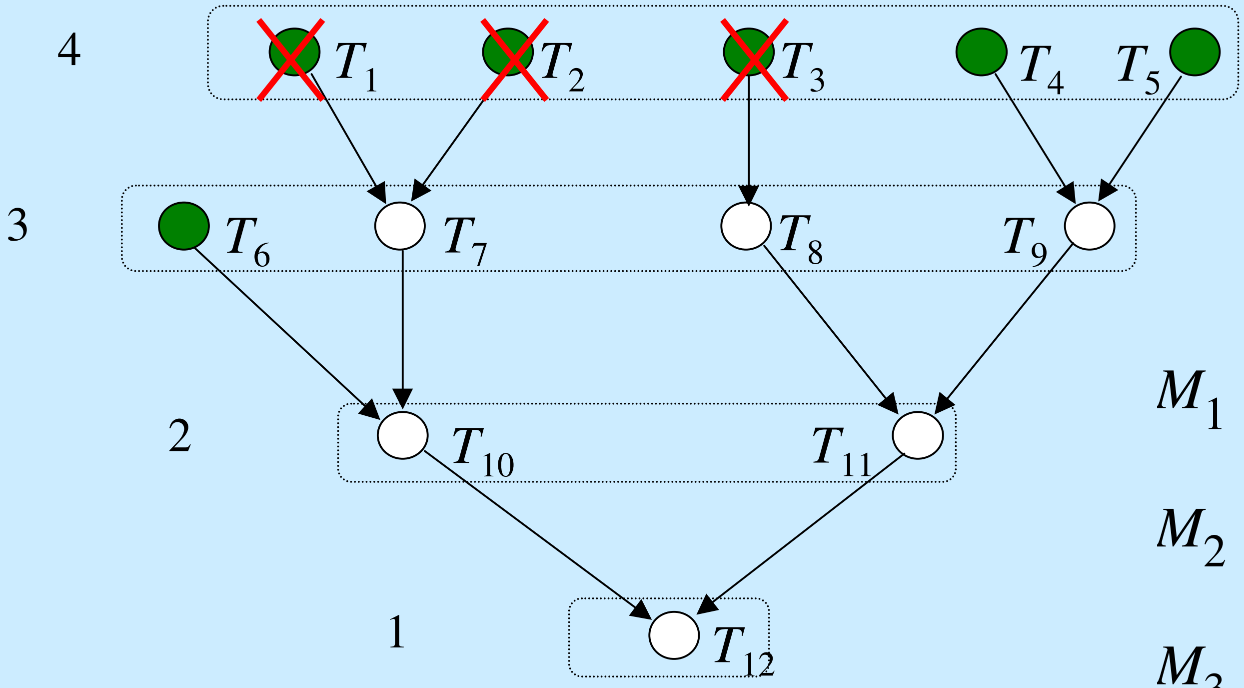
● - avalable tasks

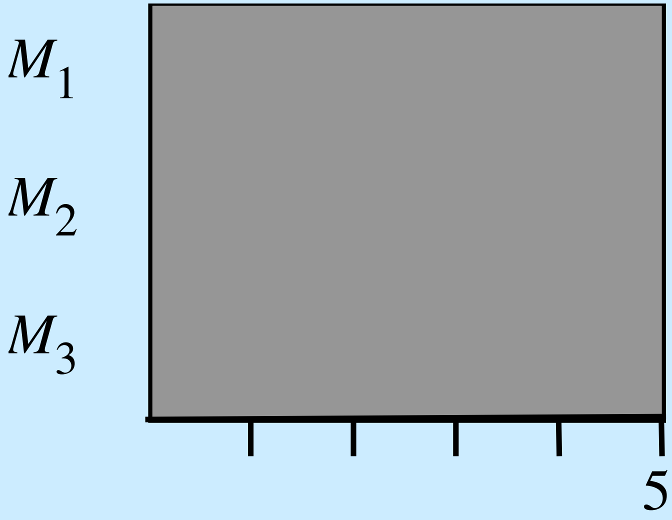
Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

Non-preemptive scheduling

Example. Hu algorithm. n=12, m=3.



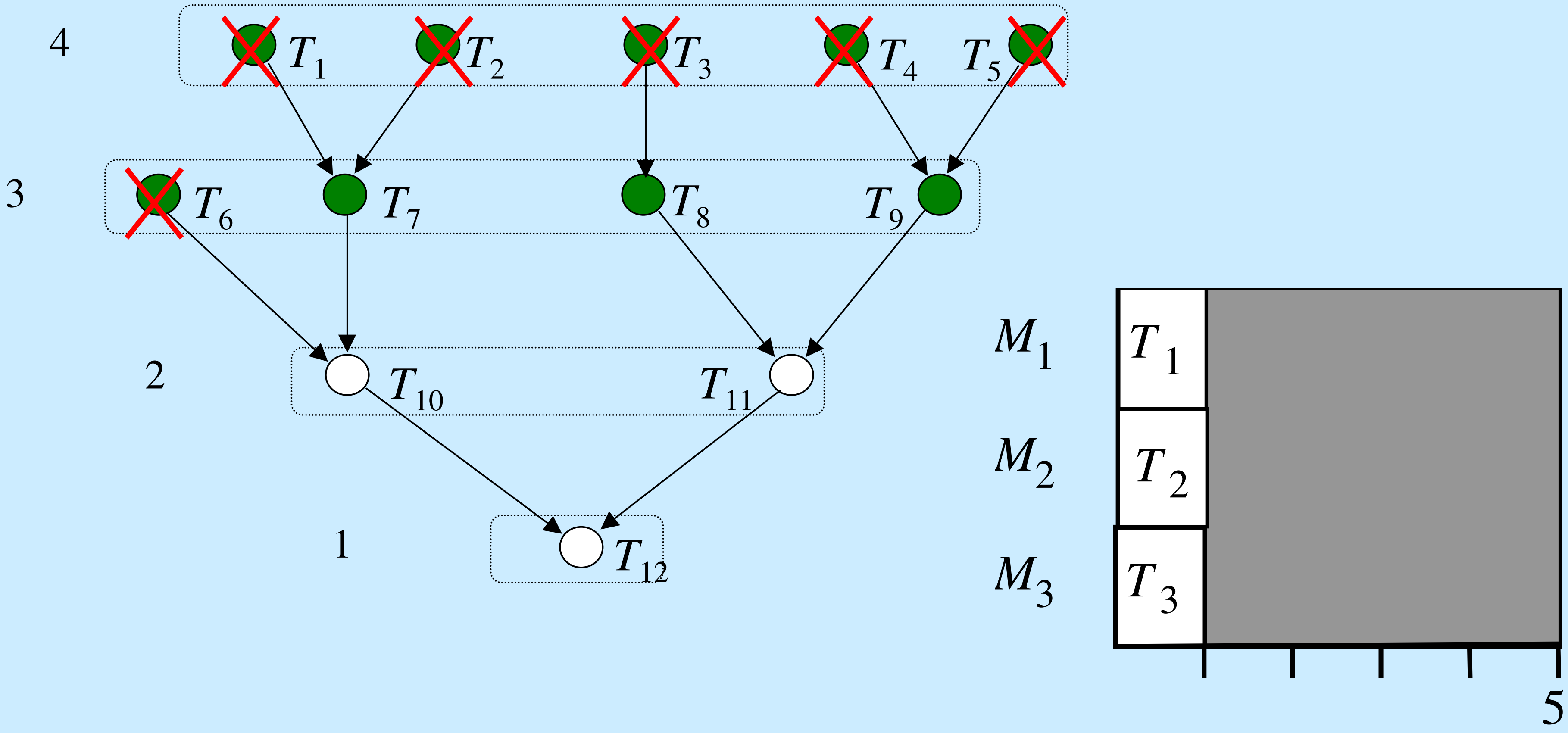


Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

Non-preemptive scheduling

Example. Hu algorithm. n=12, m=3.

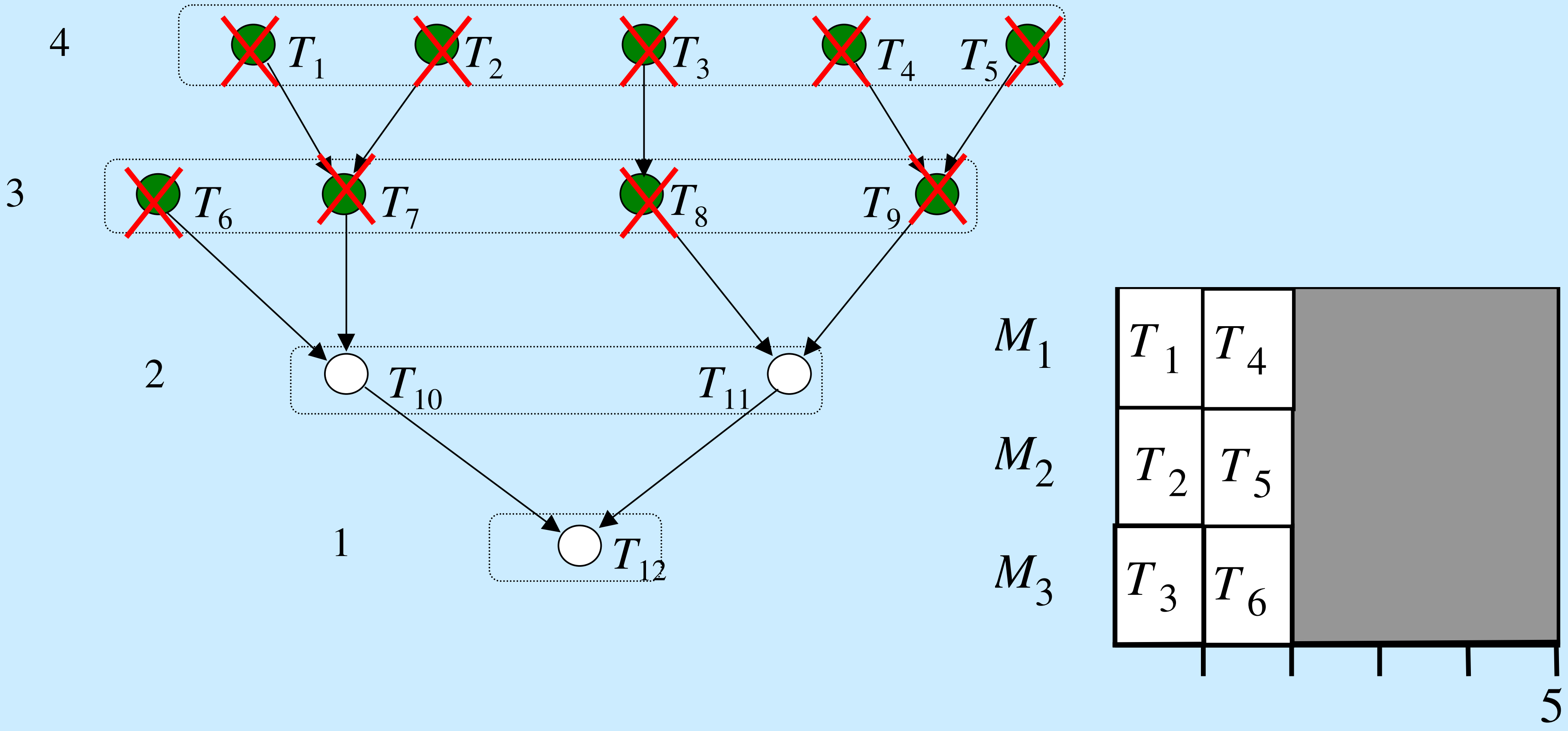


Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

Non-preemptive scheduling

Example. Hu algorithm. n=12, m=3.

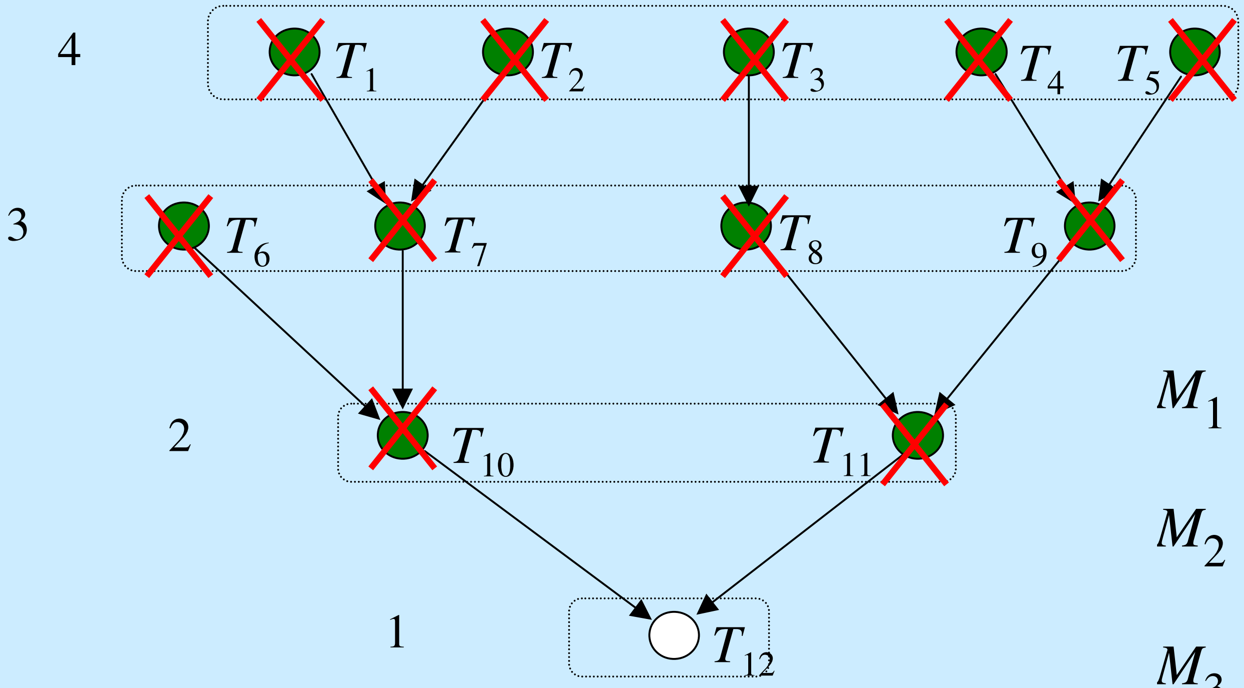


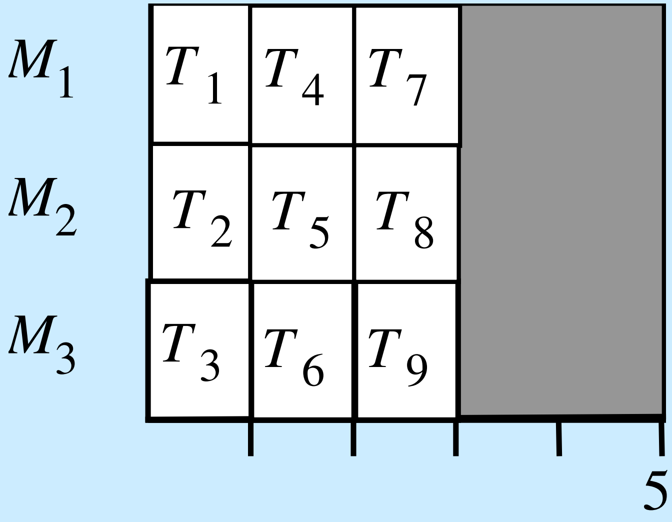
Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

Non-preemptive scheduling

Example. Hu algorithm. n=12, m=3.





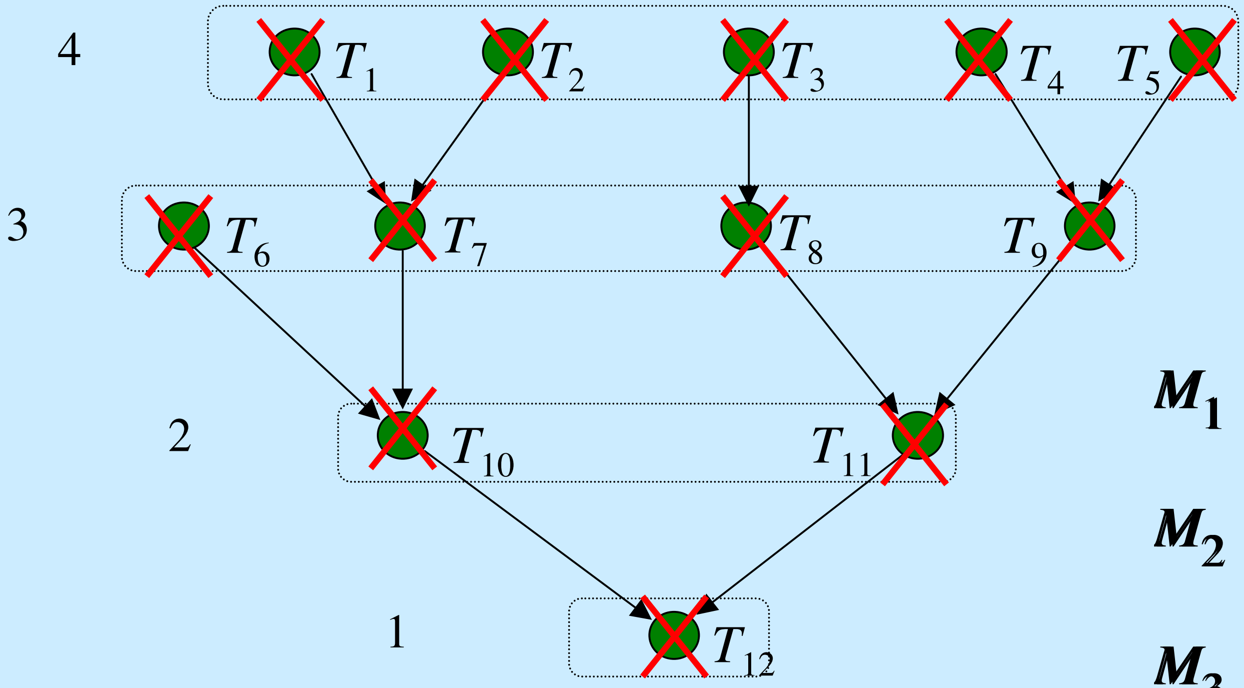
M_1	T_1	T_4	T_7
M_2	T_2	T_5	T_8
M_3	T_3	T_6	T_9

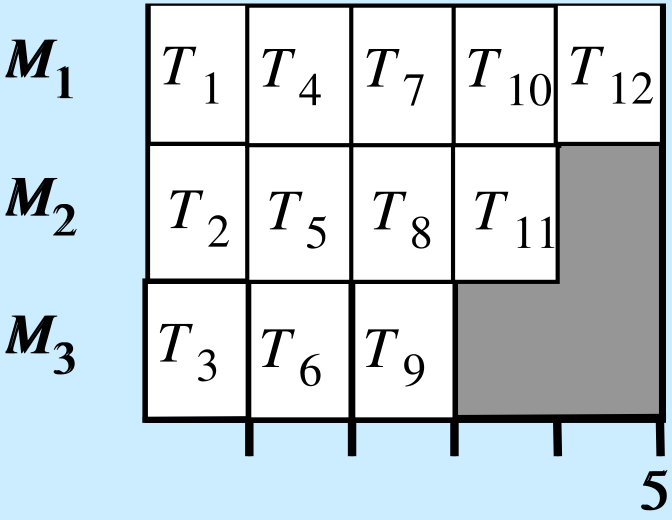
Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

Non-preemptive scheduling

Example. Hu algorithm. n=12, m=3.





M_1	T_1	T_4	T_7	T_{10}	T_{12}
M_2	T_2	T_5	T_8	T_{11}	▨
M_3	T_3	T_6	T_9	▨	▨

Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

<u>Non-preemptive scheduling</u>

Coffman-Graham algorithm (*$P2|prec|C_{max}$*):

1. label the tasks with integers *l* from range [1, ..., *n*]
2. list scheduling, with desscening labels order.

Phase 1 - task labeling;

no task has any label or list at the beginning;

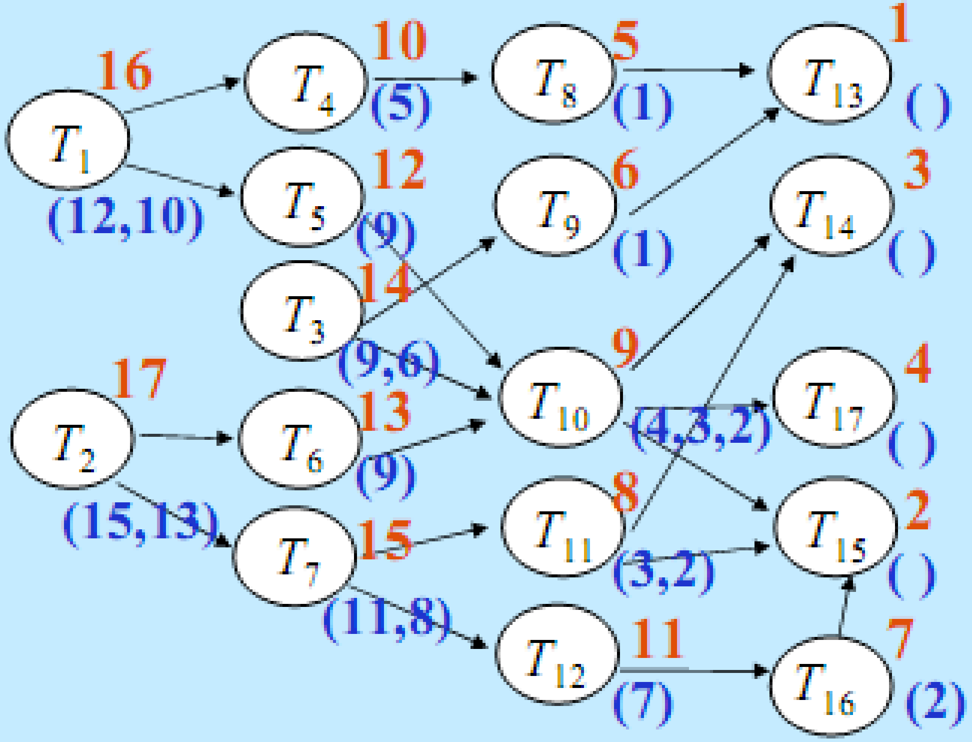
```
for $i$:=1 to $n$ do begin
    $A$:=the set of tasks without label, for which all dependent tasks are labeled;
    for each $T \in A$ do assign the descending sequence of the labels of tasks dependent of $T$ to $list(T)$;
    choose T $\in$ A with the lexicographic minimum $list(T)$;
    $l(T)$:=$i$;
end;
```

Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

Non-preemptive scheduling

Example. Coffman-Graham algorithm, $n = 17$

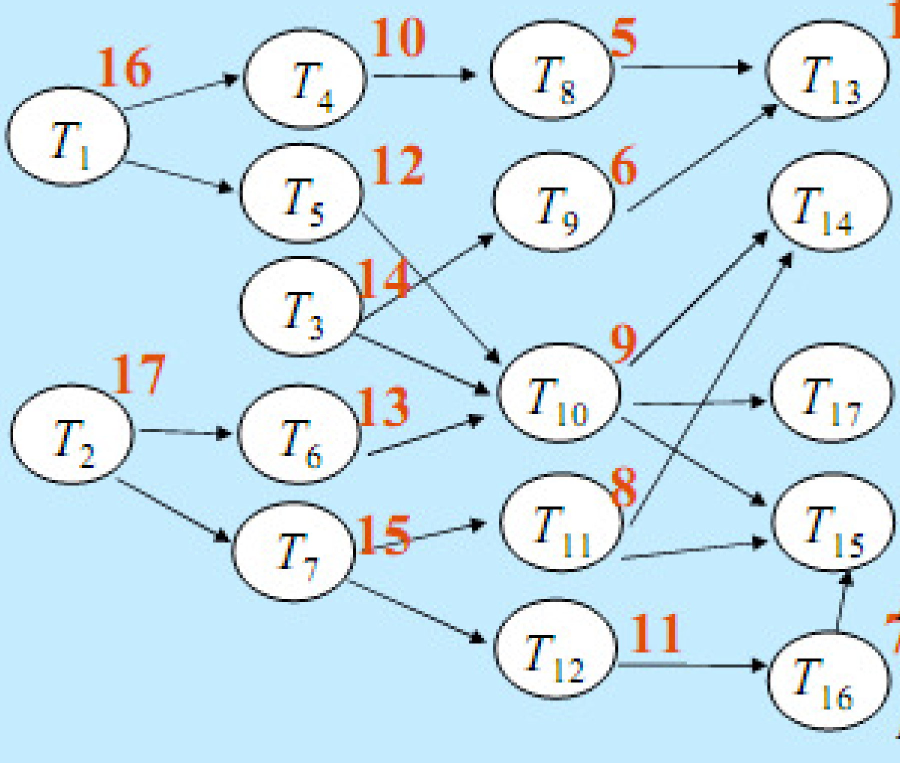


Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

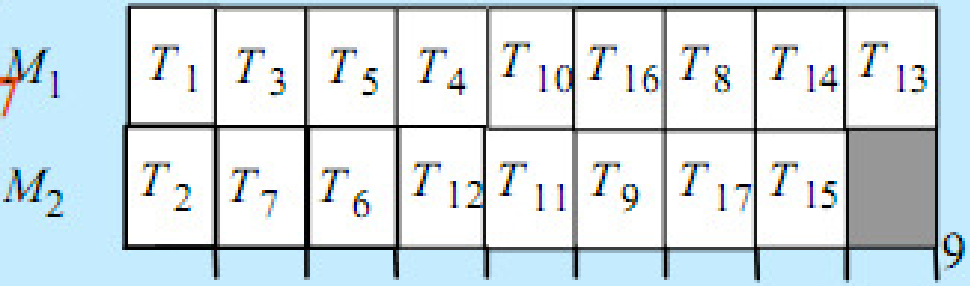
Non-preemptive scheduling

Example. Coffman-Graham algorithm, $n = 17$



The ordering in the list:

$T_2, T_1, T_7, T_3, T_6, T_5, T_{12}, T_4, T_{10}, T_{11}, T_{16}, T_9, T_8, T_{17}, T_{14}, T_{15}, T_{13}$.



M_1	T_1	T_3	T_5	T_4	T_{10}	T_{16}	T_8	T_{14}	T_{13}
M_2	T_2	T_7	T_6	T_{12}	T_{11}	T_9	T_{17}	T_{15}	▨

Scheduling on Parallel Processors to Minimize the Schedule Length.

Identical processors, dependent tasks

Non-preemptive scheduling

LS heuristic can be applied to $P|prec|C_{\max}$. The solution is 2-approximate: $C^*_{\max}(\mathrm{LS}) \le (2 - m^{-1}) C^*_{\max}$

Proof: It has been proved already...

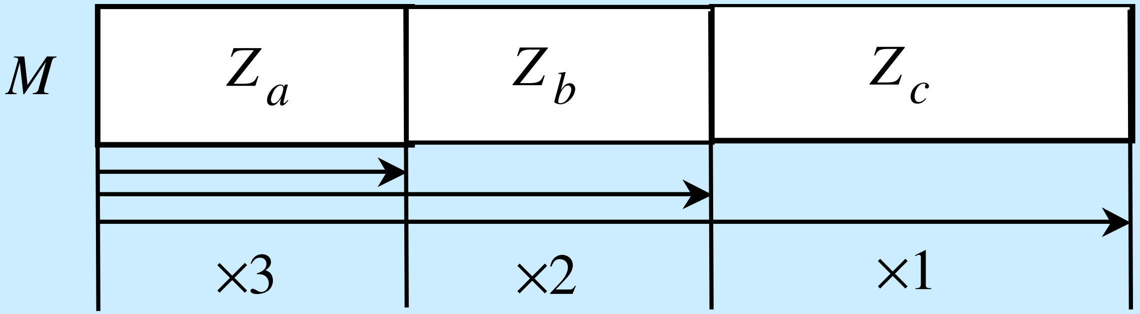
The order on the list (priority) can be chosen in many different ways. However, some anomalies may occur, i.e. the schedule may lenghten while:

- increasing the number of processors,
- decreasing the processing times,
- releasing some precedence constraints,
- changing the list order.

Scheduling on Parallel Processors to Minimize the Mean Flow Time

Identical processors, independent tasks

Proposal: task Z_j scheduled in the k-th position on machine M_i increments the value of ΣC_j by kp_j (or kp_{ij} in the case of ***R*|...**).



Corollaries.

- the processing time of the first task is multiplied by the greatest coefficient; the coefficients of the following tasks are decreasing,
- to minimize ΣC_j we should schedule short tasks first (as they are multiplied by the greatest coefficients),
- list scheduling with the ***SPT*** rule (*Shortest Processing Times*) leads to an optimal solution on a single processor,
- **how to assign the tasks to the processors?**

Scheduling on Parallel Processors to Minimize the Mean Flow Time

Identical processors, independent tasks

Both preemptive and non-preemptive cases

The problems $P||\Sigma C_i$ and $P|pmtn|\Sigma C_i$ can be considered together (preemptions do not improve the criterion).

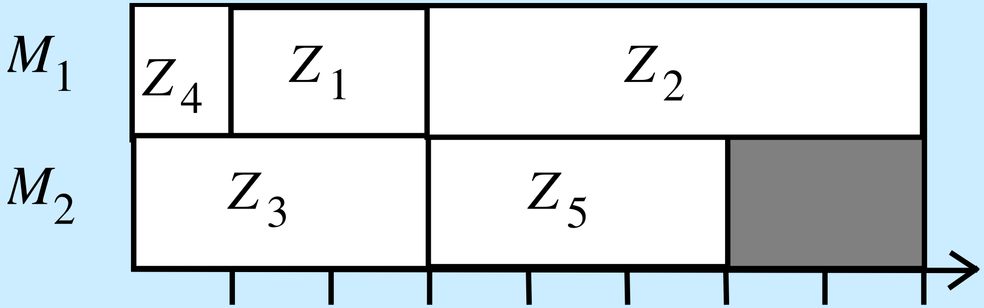
Optimal algorithm ***O*(*n*log *n*)**:

1. Suppose the number of tasks is a multiplicity of m (introduce empty tasks if needed),
2. Sort the tasks according to ***SPT***,
3. Assing the following *m*–tuples of tasks to the processors arbitrarily.

Example. m=2, n=5, $p_1,\ldots,p_5$=2,5,3,1,3.

SPT:	Z_0	Z_4	Z_1	Z_3	Z_5	Z_2
p_i=	0	1	2	3	3	5

Empty task



ΣC_j*=21

Scheduling on Parallel Processors to Minimize the Mean Flow Time

Identical processors, independent tasks

Both preemptive and non-preemptive cases

The problems $P||\Sigma C_i$ and $P|pmtn|\Sigma C_i$ can be considered together (preemptions do not improve the criterion).

Optimal algorithm ***O*(*n*log *n*)**:

1. Suppose the number of tasks is a multiplicity of m (introduce empty tasks if needed),
2. Sort the tasks according to ***SPT***,
3. Assing the following *m*–tuples of tasks to the processors arbitrarily.

Proof (the case of non-preemptive scheduling):

Lemma. Suppose $a_1,...,a_n$ i $b_1,...,b_n$ are sequences of positive integers. How to permutate them in order to make the dot product

$$a_{\pi(1)}b_{\pi(1)}+a_{\pi(2)}b_{\pi(2)}+...+a_{\pi(n-1)}b_{\pi(n-1)}+a_{\pi(n)}b_{\pi(n)}$$

- the greatest possible? – both should be sorted in ascending order,
- the smallest possible? – sort one ascending, the second descending

Scheduling on Parallel Processors to Minimize the Mean Flow Time

Identical processors, independent tasks

Both preemptive and non-preemptive cases

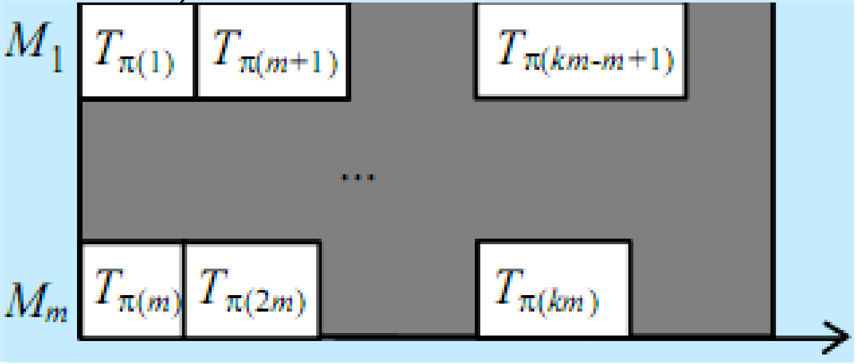
The problems $P||\Sigma C_i$ and $P|pmtn|\Sigma C_i$ can be considered together (preemptions do not improve the criterion).

Optimal algorithm $\boldsymbol{O(n\log n)}$:

1. Suppose the number of tasks is a multiplicity of m (introduce empty tasks if needed),
2. Sort the tasks according to ***SPT***,
3. Assing the following *m*–tuples of tasks to the processors arbitrarily.

Proof (the case of non-preemptive scheduling). Consider an optimal scheduling. One may assume that there are k tasks scheduled on each processors (introducing empty tasks if needed).

$$\Sigma C_i = kp_{\pi(1)}+\ldots+kp_{\pi(m)}+$$
$$+(k-1)p_{\pi(m+1)}+\ldots+(k-1)p_{\pi(2m)}+$$
$$+1p_{\pi(km-m+1)}+\ldots+1p_{\pi(km)}$$



Reordering the tasks according to the SPT rule does not increase ΣC_i

Scheduling on Parallel Processors to Minimize the Mean Flow Time

Identical processors, independent tasks

Non-preemptive scheduling

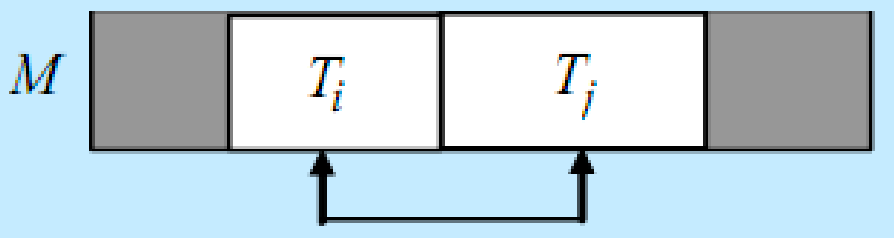
In the case the weights are introduced, even $\boldsymbol{P2||\Sigma w_jC_j}$ is NP-hard.

Proof (sketch). Similar to $\boldsymbol{P2||C_{\max}}$. PP ➔ $\boldsymbol{P2||\Sigma w_iC_i}$ reduction: take n tasks with $p_j = w_j = a_j$ ($j=1,\dots,n$), two processors. There exists a number $C(a_1,\dots,a_n)$ such that $\Sigma w_jC_j \le C(a_1,\dots,a_n) \Leftrightarrow C_{\max}{}^* = \Sigma_{i=1,\dots,n}\, a_i/2$ (exercise).

Following ***Smith rule*** in the case of single processor scheduling $\boldsymbol{1||\Sigma w_jC_j}$ leads to an optimal solution in $\boldsymbol{O(n\log n)}$ time:

- sort the tasks in ascending p_j/w_j order.

Proof. Consider the improvement of the criterion caused by changing two consecutive tasks.



$$w_jp_j + w_j(p_j+p_j) - p_iw_i - w_j(p_j+p_j) =$$
$$= w_ip_j - p_jw_i \ge 0 \Leftrightarrow p_j/w_j \ge p_i/w_i$$

violating Smith rule increases $\boldsymbol{\Sigma w_jC_j}$

Scheduling on Parallel Processors to Minimize the Mean Flow Time

Non-preemptive scheduling

RPT rule can be used in order to minimize both $C_{\max}$ and ΣC_i:

1. Use the LPT algorithm.
2. Reorder the tasks within each processor due to the SPT rule.
   Approximation ratio: $1 \leq \Sigma C_i^{\,(\mathrm{RPT})} / \Sigma C_i^* \leq m$ (commonly better)

Identical processors, dependent tasks

- Even $P|prec,p_j=1|\Sigma C_i$, $P2|prec,p_i \in \{1,2\}|\Sigma C_i$, $P2|chains|\Sigma C_i$ and $P2|chains,pmtn|\Sigma C_i$ are NP-hard.
- Polynomial-time algorithms solving $P|prec,p_j=1|\Sigma C_i$ (Coffman-Graham) and $P|out\text{-}tree,p_j=1|\Sigma C_i$ (adaptation of Hu algorithm) are known.
- In the case of weighted tasks even single machine scheduling of unit time tasks $1|prec,p_j=1|\Sigma w_i C_i$ is NP-hard.

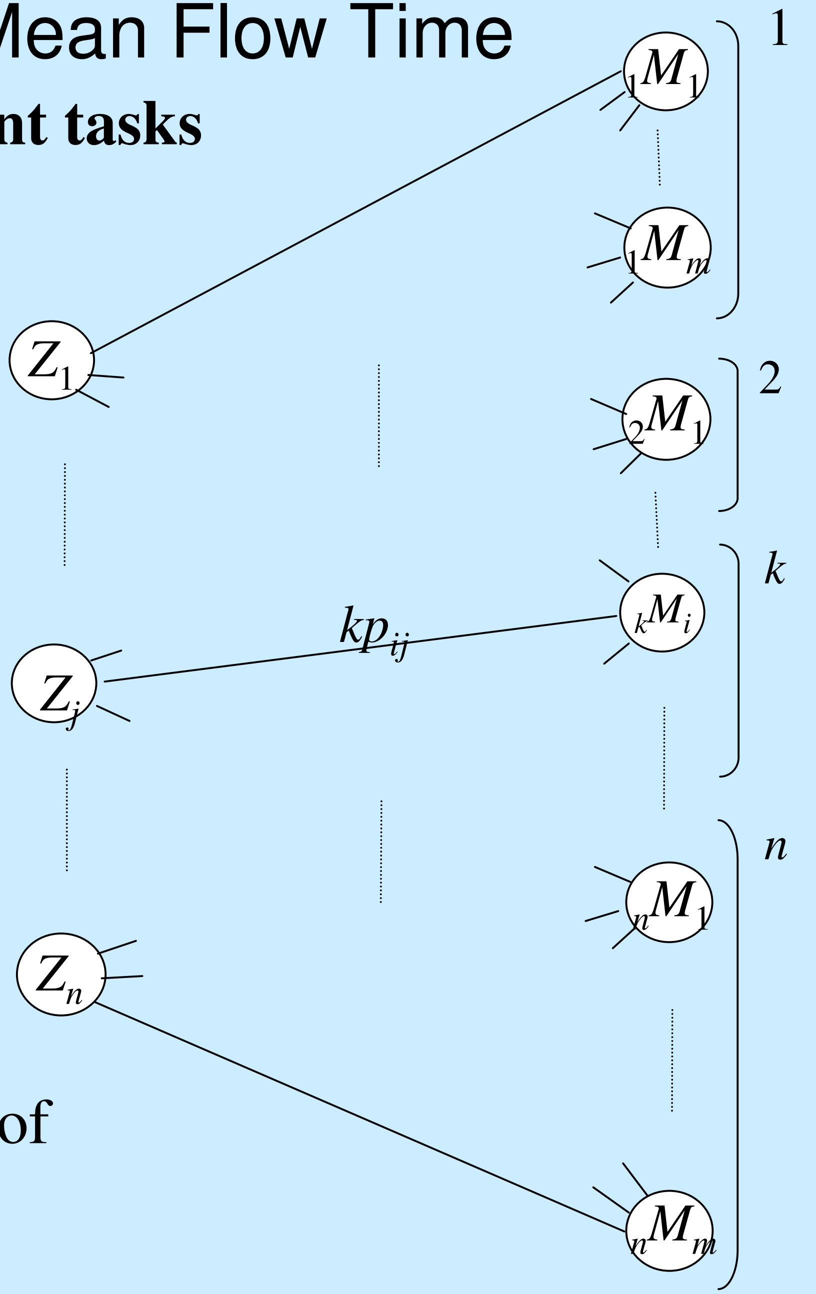
Scheduling on Parallel Processors to Minimize the Mean Flow Time

Unrelated processors, independent tasks

$O(n^3)$ algorithm for $R||\Sigma C_i$ is based on the problem of graph matching.

Bipartite weighted graph:

- Partition V_1 corresponding to the tasks $Z_1,...,Z_n$.
- Partition V_2 – each processor n times: ${}_kM_i$, $i=1...m$, $k=1...n$.
- The edge connecting Z_j and ${}_kM_i$ is weighted with kp_{ij} (it corresponds to scheduling task Z_j on M_i, in the k-th position from the end).

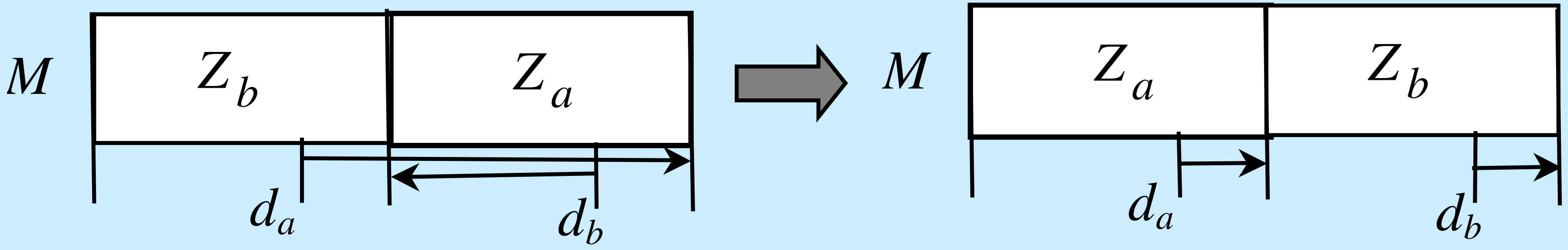


We construct the lightest matching of n edges, which corresponds to optimal scheduling.

Scheduling on Parallel Processors to Minimize the Maximum Lateness

Properties:

•$L_{\max}$ criterion is a generalization of $C_{\max}$, the problems that are NP–hard in the case of minimizing $C_{\max}$ remain NP-hard in the case of $L_{\max}$,



- if we have several tasks of different due times we should start with the most urgent one to minimize maximum lateness,
- this leads to ***EDD rule*** (*Earliest Due Date*) – choose tasks in the ordering of ascending due dates d_j,
- the problem of scheduling on a single processor (**1||*L*$_{\max}$**) is solved by using the ***EDD*** rule.

Scheduling on Parallel Processors to Minimize the Maximum Lateness

Identical processors, independent tasks

Preemptive scheduling

Single machine: ***Liu algorithm*** $\boldsymbol{O(n^2)}$**,** based on the *EDD* rule, solves $\mathbf{1}|r_i,pmtn|L_{\max}$:

1. Choose the task of the smallest due date among available ones,
2. Every time a task has been completed or a new task has arrived go to 1. (in the latter case we preempt currently processed task).

Arbitrary number of processors($P|r_i,pmtn|L_{\max}$). A polynomial-time algorithm is known:

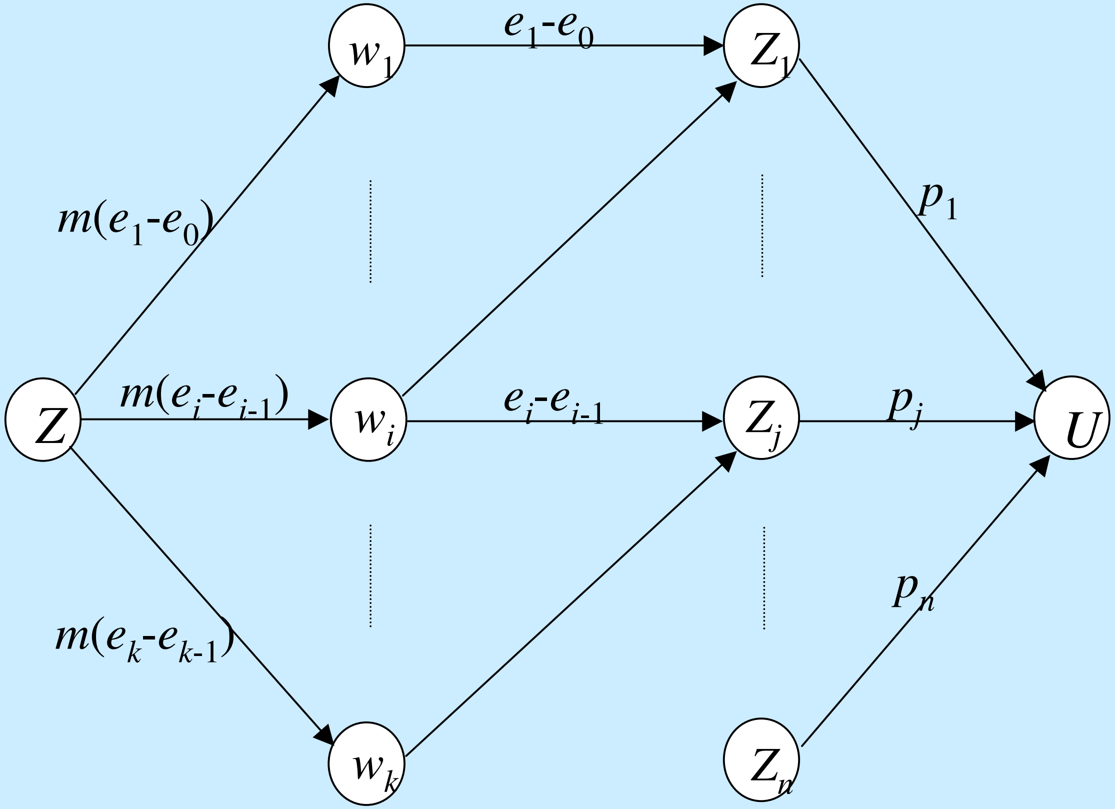
We use sub-routine solving the problem with deadlines $P|r_i,C_i\le d_i,pmtn|–$, We find the optimal value of $L_{\max}$ using binary search algorithm.

Scheduling on Parallel Processors to Minimize the Maximum Lateness

$P|r_i,C_i\leq d_i,pmtn|-$ to the problem of maximum flow. First we put the values r_i and d_i into the ascending sequence $e_0<e_1<\ldots<e_k$.

We construct a network:

- The source is connected by k arc of capacity $m(e_i-e_{i-1})$ to nodes w_i, $i=1,\ldots,k$.
- The arcs of capacity p_i connect nodes-tasks Z_i, to the sink; $i=1,\ldots,n$.
- We connect w_i and Z_j by an arc of capacity e_i-e_{i-1}, iff $[e_{i-1},e_i]\subset[r_j,d_j]$.



A schedule exists ⇔ there exists a flow of value $\Sigma_{i=1,\ldots,n}\, p_i$ (one can distribute the processors among tasks in the time intervals $[e_{i-1}, e_i]$ to complete all the tasks).

Scheduling on Parallel Processors to Minimize the Maximum Lateness

Dependent tasks

Non-preemptive scheduling

Some NP-hard cases: $P2||L_{\max}$, $1|r_j|L_{\max}$.

Polynomial-time solvable cases:

- unit processing times $P|p_j=1,r_j|L_{\max}$.
- similarly for uniform processors $Q|p_j=1|L_{\max}$ (the problem can be reduced to linear programming),
- single processor scheduling $1||L_{\max}$ – solvable using the EDD rule (has been discussed already...).

Scheduling on Parallel Processors to Minimize the Maximum Lateness

Dependent tasks

Preemptive scheduling

Single processor scheduling **$1|pmtn,prec,r_j|L_{\max}$** can be solved by a modified Liu algorithm $O(n^2)$:

1. Determine ***modified due dates*** for each task:

$$d_j^*=\min\{d_j, \min\{d_i: Z_j \prec Z_i\}\}$$

2. Apply the EDD rule using d_j^* values, preempting current task in the case a task with smaller modified due-date gets available,
3. Repeat 2 until all the tasks are completed.

- Some other polynomial-time cases:

  $P|pmtn,in\text{–}tree|L_{\max}$, $Q2|pmtn,prec,r_j|L_{\max}$.

- Moreover, some pseudo-polynomial time algorithms are known.

Scheduling on Parallel Processors to Minimize the Maximum Lateness

Dependent tasks

Non-preemptive scheduling

- Even $P|p_j=1,out\text{–}tree|L_{\max}$ is NP–hard.

A polynomial-time algorithm for $P2|prec,p_j=1|L_{\max}$ is known.

- $P|p_j=1,in\text{–}tree|L_{\max}$ can be solved by ***Brucker algorithm*** $O(n\log n)$:

$next(j)$ = immediate successor of task T_j.

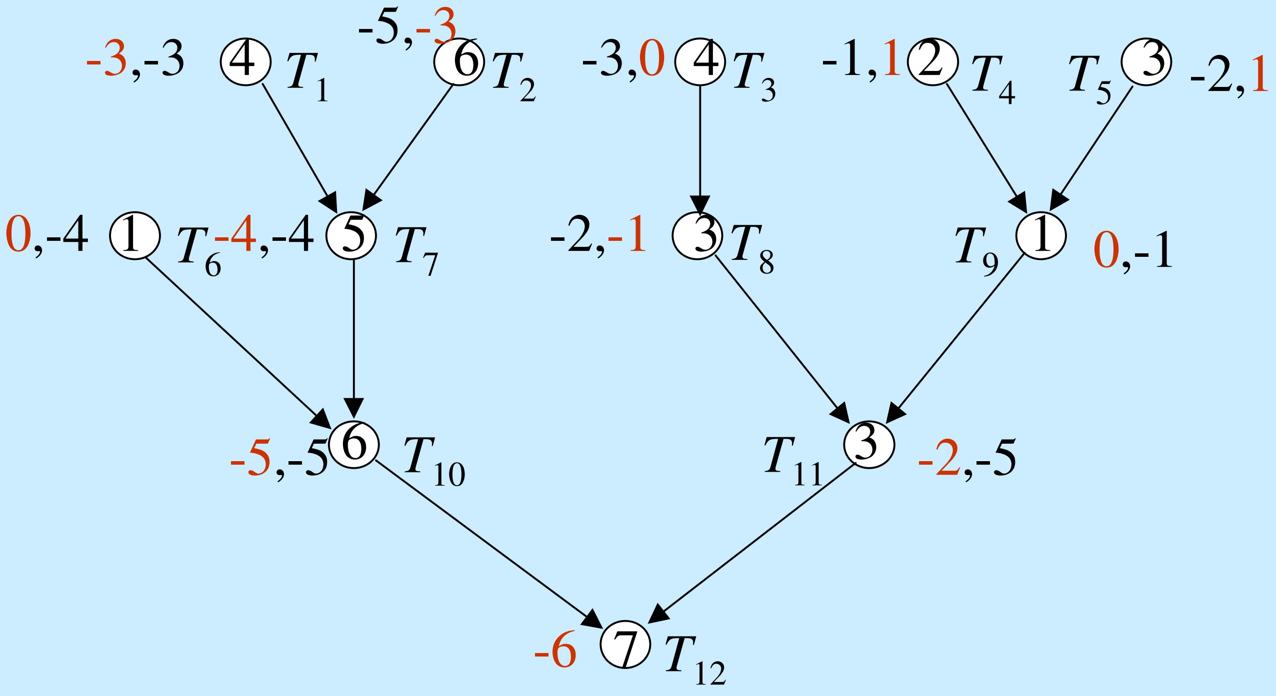
1. Derive *modified due dates*: $d_{\mathrm{root}}{}^*=1-d_{\mathrm{root}}$ for the root and $d_k{}^*=\max\{1+d_{next(k)}{}^*,1-d_k\}$ for other tasks,
2. Schedule the tasks in a similar way as in Hu algorithm, choosing every time the tasks of the largest modified due date instead of the largest distance from the root.

Scheduling on Parallel Processors to Minimize the Maximum Lateness

Dependent tasks

Non-preemptive scheduling

Example. Brucker algorithm, n=12, m=3, due dates in the nodes.

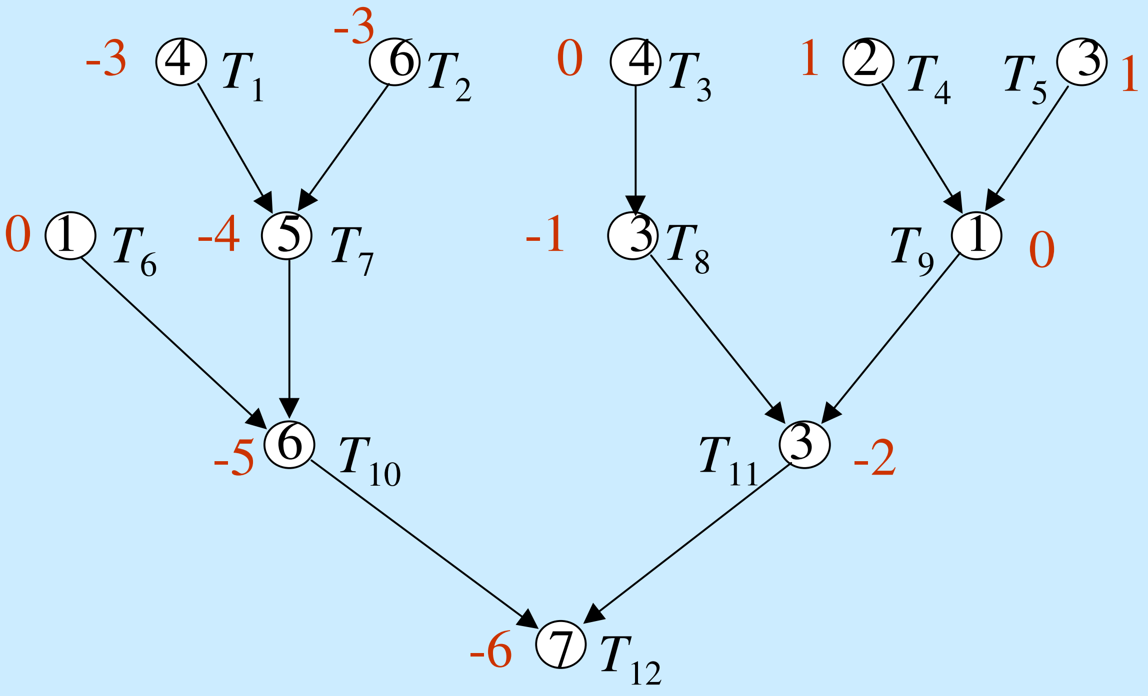


Scheduling on Parallel Processors to Minimize the Maximum Lateness

Dependent tasks

Non-preemptive scheduling

Example. Brucker algorithm, n=12, m=3, due dates in the nodes

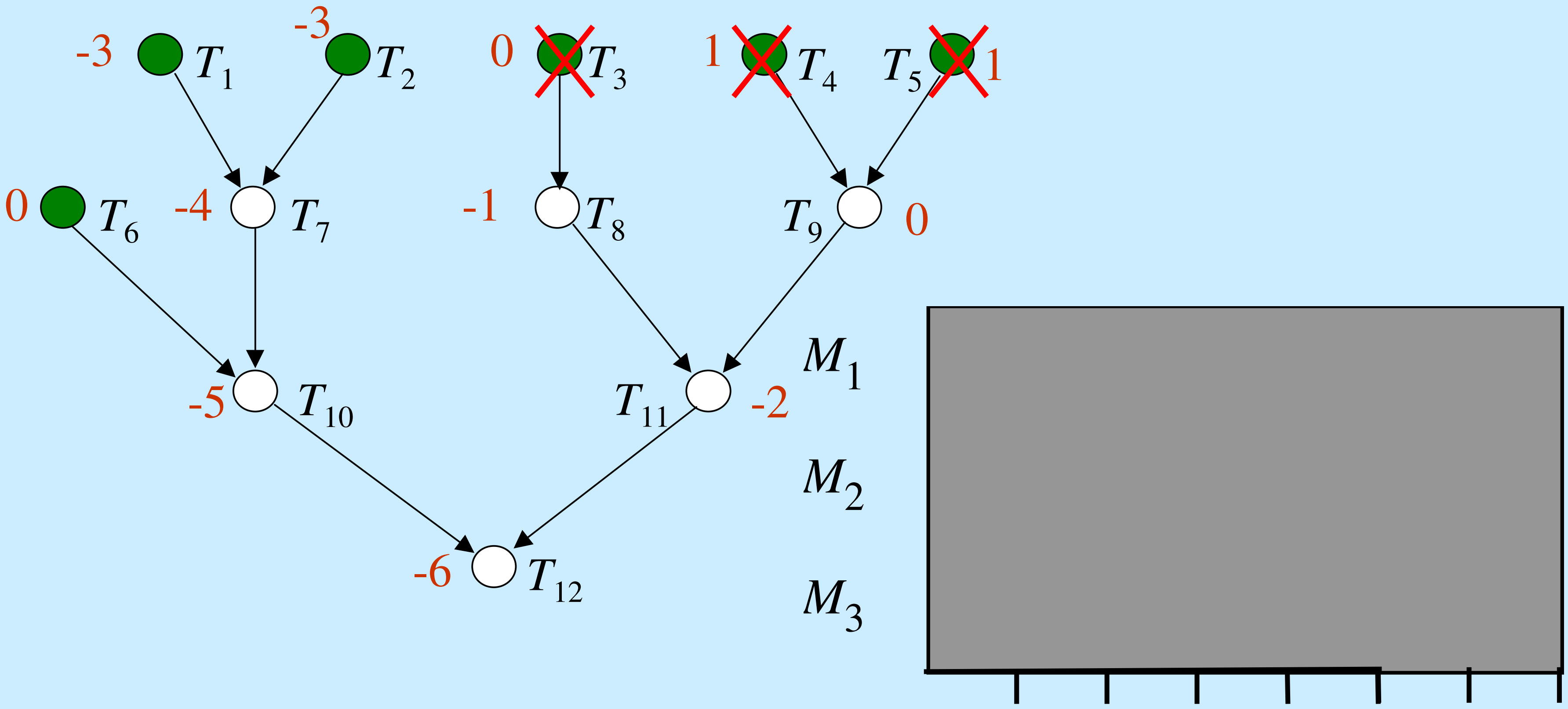


Scheduling on Parallel Processors to Minimize the Maximum Lateness

Dependent tasks

Non-preemptive scheduling

Example. Brucker algorithm, n=12, m=3, due dates in the nodes

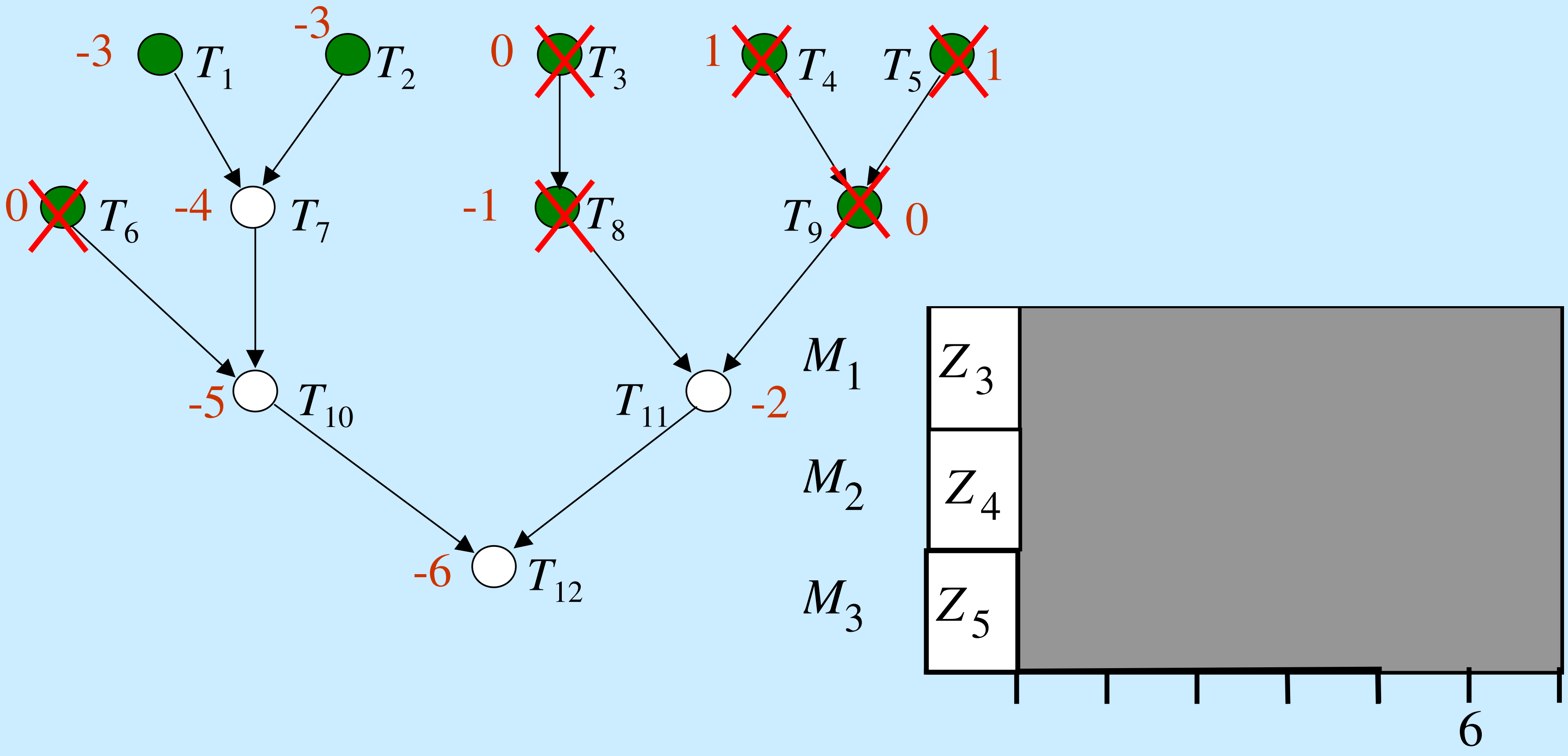


Scheduling on Parallel Processors to Minimize the Maximum Lateness

Dependent tasks

Non-preemptive scheduling

Example. Brucker algorithm, n=12, m=3, due dates in the nodes

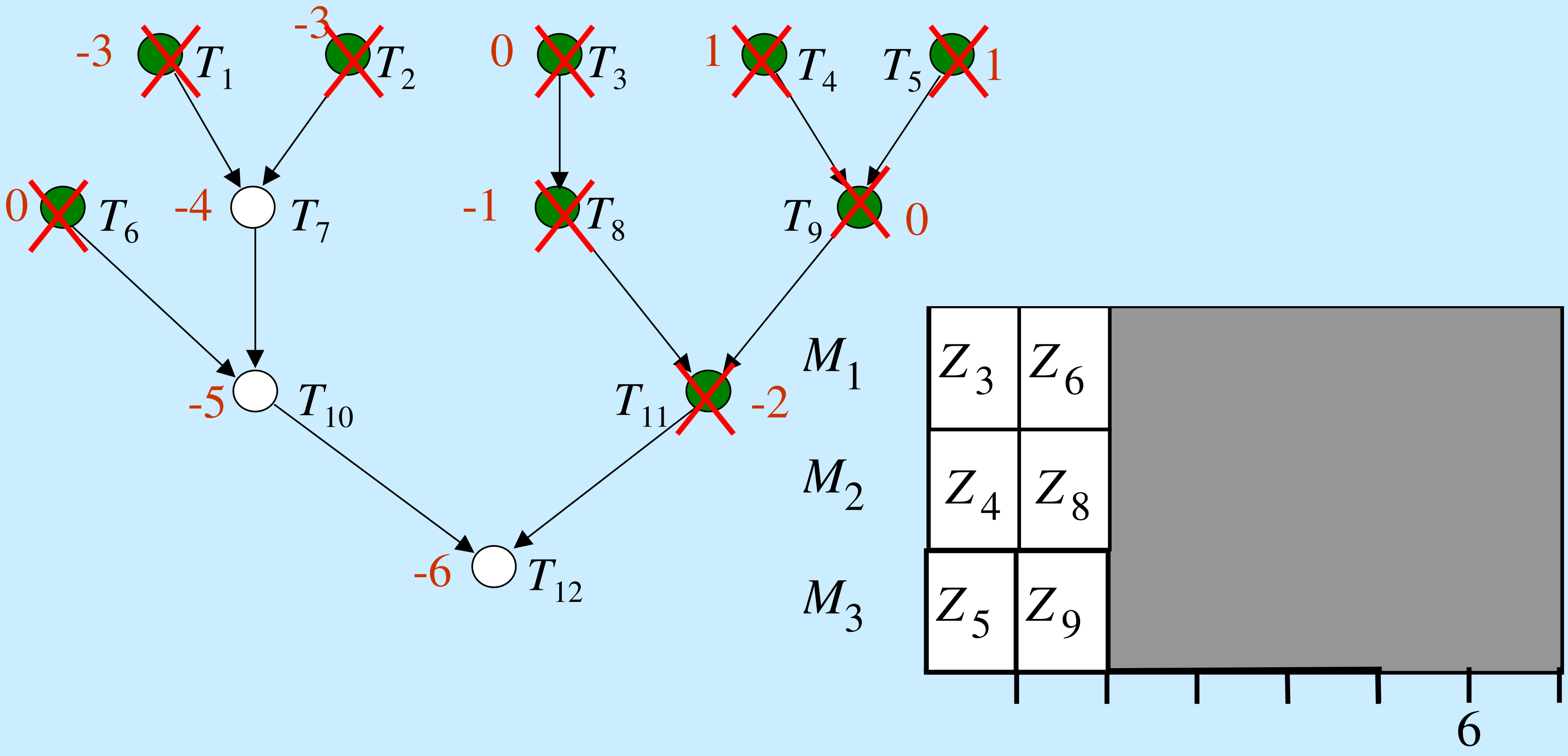


Scheduling on Parallel Processors
to Minimize the Maximum Lateness

Dependent tasks

Non-preemptive scheduling

Example. Brucker algorithm, n=12, m=3, due dates in the nodes

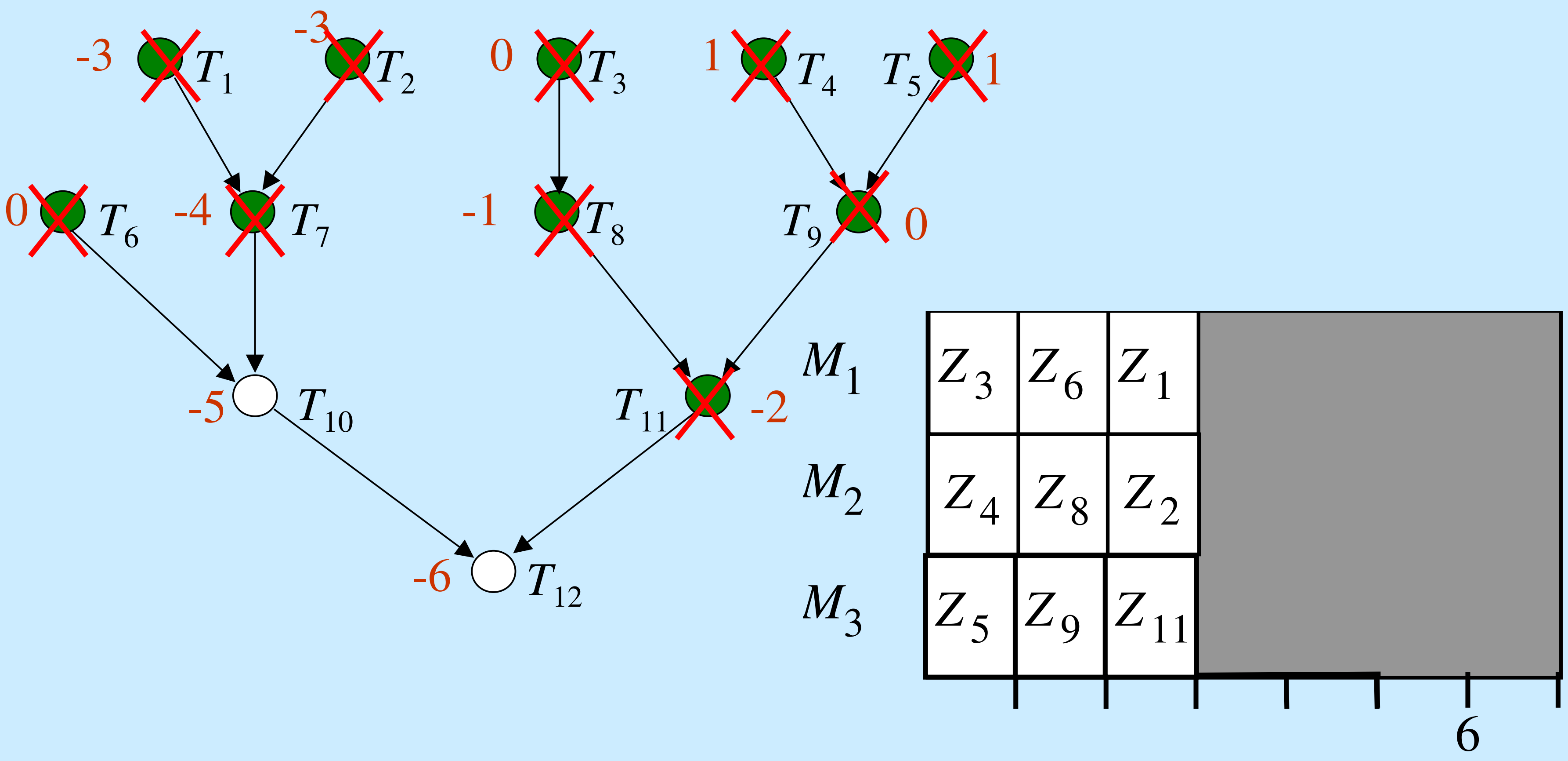


Scheduling on Parallel Processors to Minimize the Maximum Lateness

Dependent tasks

Non-preemptive scheduling

Example. Brucker algorithm, n=12, m=3, due dates in the nodes



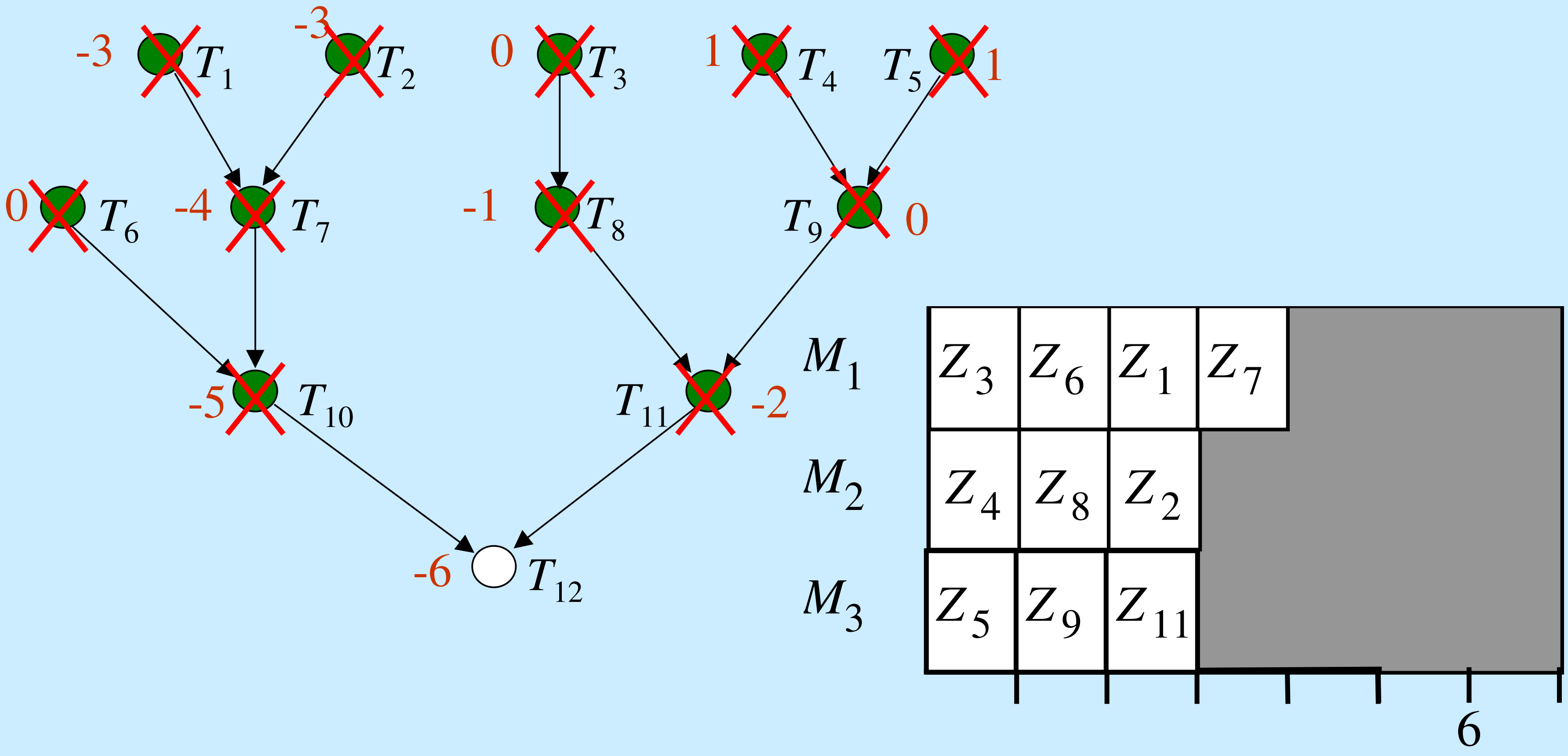
M_1	Z_3	Z_6	Z_1
M_2	Z_4	Z_8	Z_2
M_3	Z_5	Z_9	Z_{11}

Scheduling on Parallel Processors to Minimize the Maximum Lateness

Dependent tasks

<u>Non-preemptive scheduling</u>

Example. Brucker algorithm, n=12, m=3, due dates in the nodes



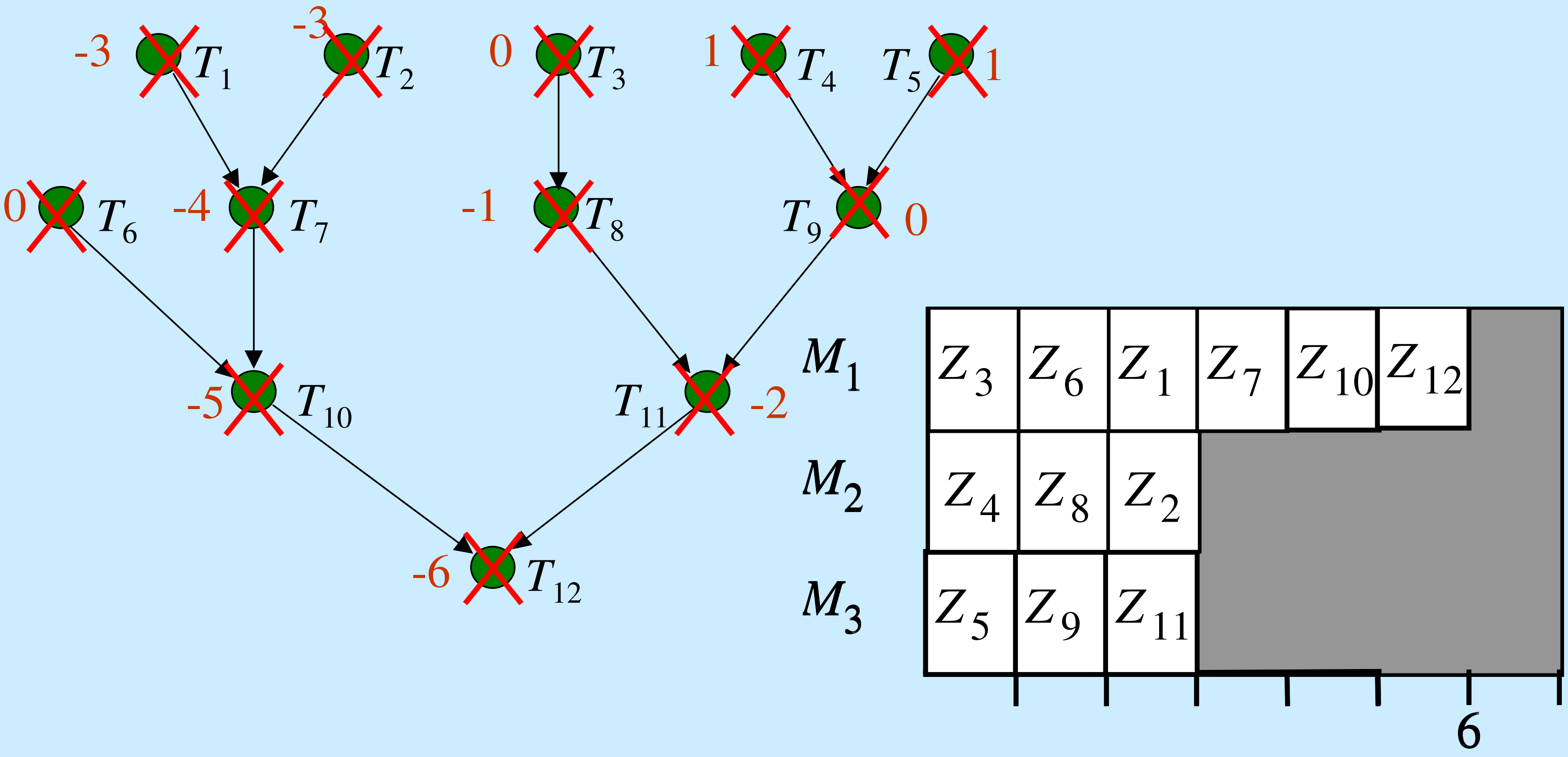
M_1	Z_3	Z_6	Z_1	Z_7
M_2	Z_4	Z_8	Z_2	
M_3	Z_5	Z_9	Z_{11}	

Scheduling on Parallel Processors to Minimize the Maximum Lateness

Dependent tasks

Non-preemptive scheduling

Example. Brucker algorithm, n=12, m=3, due dates in the nodes



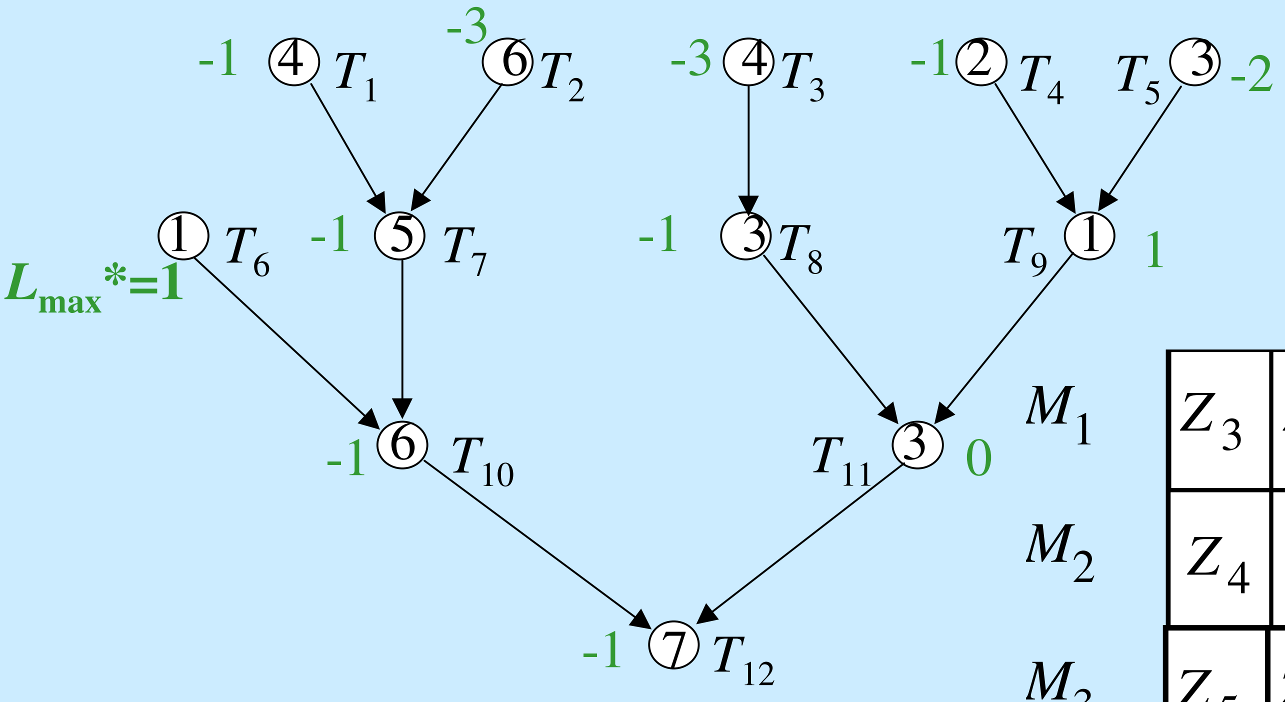
Scheduling on Parallel Processors to Minimize the Maximum Lateness

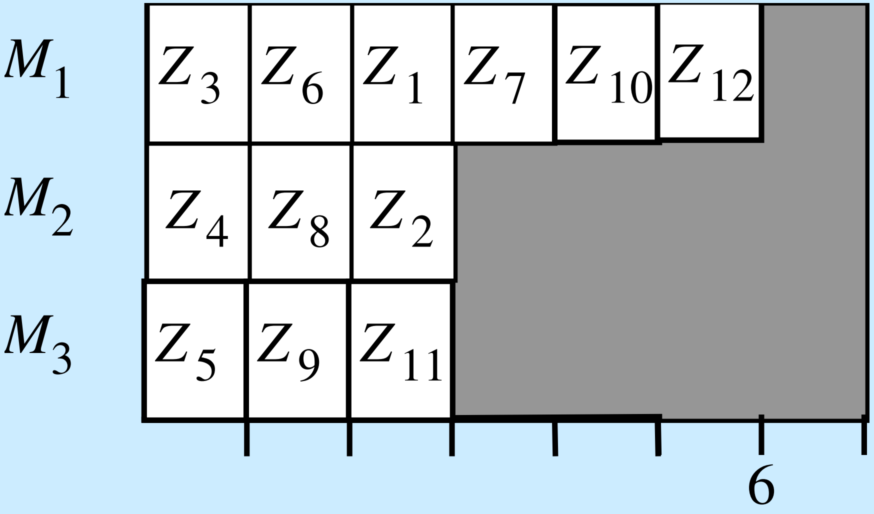
Dependent tasks

Non-preemptive scheduling

Example. Brucker algorithm, n=12, m=3, due dates in the nodes

Lateness:





M_1	Z_3	Z_6	Z_1	Z_7	Z_{10}	Z_{12}
M_2	Z_4	Z_8	Z_2			
M_3	Z_5	Z_9	Z_{11}			

Minimizing the Number of Tardy Tasks on a Single Machine

Independent non-preemptive tasks

Obviously even $P2||\Sigma U_i$ and $P2||\Sigma T_i$ are NP-hard.

Proof. Similar to $P2||C_{\max}$.

Further we consider single processor scheduling only.

Minimizing the Number of Tardy Tasks on a Single Machine

Independent non-preemptive tasks

Obviously even $P2||\Sigma U_i$ and $P2||\Sigma T_i$ are NP-hard.

Proof. Similar to $P2||C_{\max}$.

Further we consider single processor scheduling only.

Minimizing number of tardy tasks $1||\Sigma U_i$ is polynomial-time solvable

Hodgson algorithm $O(n\log n)$:

```
Sort the tasks with the EDD rule: $T_{\pi(1)}, T_{\pi(2)}, \ldots, T_{\pi(n)}$;
$A := \emptyset$;
for $i := 1$ to $n$ do begin
    $A := A \cup \{T_{\pi(i)}\}$;
    if $\Sigma_{Zj\in A}\, p_j > d_{\pi(i)}$ then remove the longest task from $A$;
end;
```

A – maximum cardinality set of tasks that can be scheduled on time.

Minimizing the Number of Tardy Tasks on a Single Machine

Independent non-preemptive tasks

Obviously even $P2||\Sigma U_i$ and $P2||\Sigma T_i$ are NP-hard.

Proof. Similar to $P2||C_{\max}$.

Further we consider single processor scheduling only.

Minimizing number of tardy tasks $1||\Sigma U_i$ is polynomial-time solvable

Hodgson algorithm $O(n\log n)$:

```
Sort the tasks with the EDD rule: $T_{\pi(1)}, T_{\pi(2)}, \dots, T_{\pi(n)}$;
$A := \varnothing$;
for $i := 1$ to $n$ do begin
    $A := A \cup \{T_{\pi(i)}\}$;
    if $\Sigma_{Z_j \in A}\, p_j > d_{\pi(i)}$ then remove the longest task from $A$;
end;
Schedule $A$ in the EDD order, then the rest of the tasks in any order;
```

A – maximum cardinality set of tasks that can be scheduled on time.

Minimizing the Number of Tardy Tasks on a Single Machine

Independent non-preemptive tasks

- Weighted tasks scheduling $1||\Sigma w_i U_i$ is NP–hard as a generalization of knapsack problem (the same for $1||\Sigma w_i T_i$) .
- Assuming unit execution times makes these problems easier: $1|p_j=1|\Sigma w_i U_i$ i $1|p_j=1|\Sigma w_i T_i$ are polynomial time solvable – a natural reduction to the problem of lightest matching in a bipartite graph.

Minimizing the Number of Tardy Tasks on a Single Machine

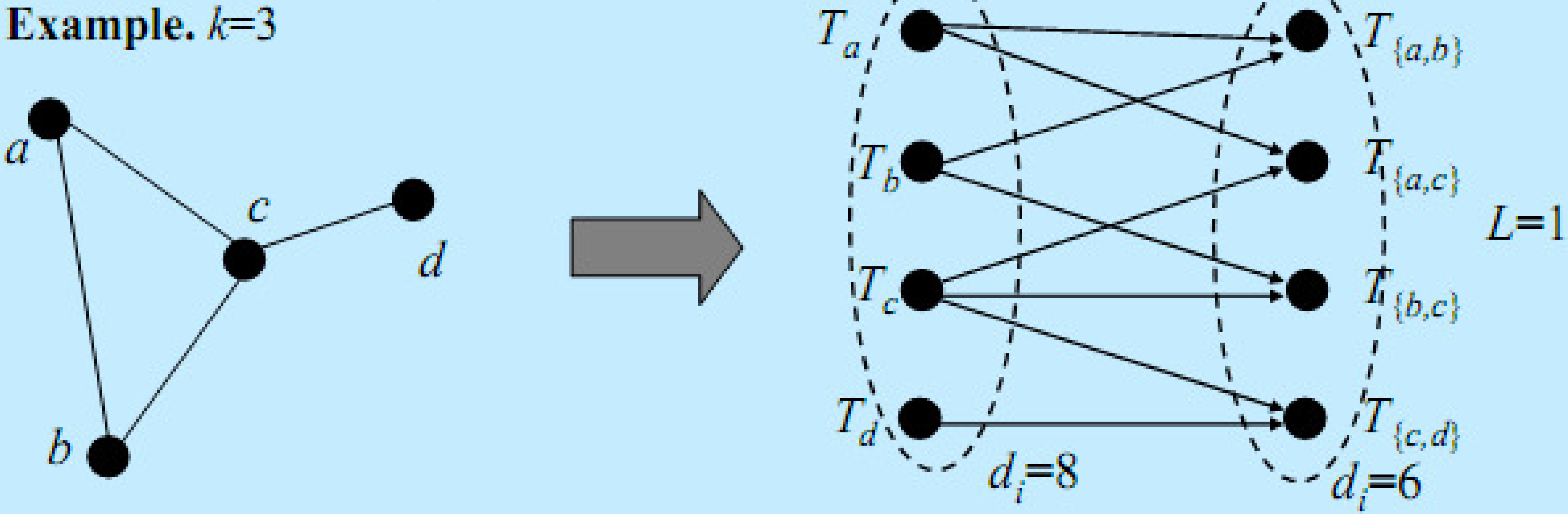
Dependent and non-preemptive tasks

NP-hardness even with unit processing times for the problems $1|p_j\!=\!1, prec|\Sigma U_i$ and $1|p_j\!=\!1, prec|\Sigma T_i$.

Proof. *Clique problem*: for a given graph $G\,(V,E)$ and a positive integer k determine if G contains a complete subgraph of k vertices.

CP ➔ $1|p_j\!=\!1, prec|\Sigma U_i$ reduction: We put unit processing time tasks T_v with $d_v\!=\!|V\cup E|$ for every vertex $v\in V$ and T_e with $d_e\!=\!k+k(k-1)/2$ for every edge $e\in E$. Precedence constraints: $T_v \prec T_e \Leftrightarrow v$ is incident to e. Limit $L\!=\!|E|-k(k-1)/2$.

Example. k=3



Minimizing the Number of Tardy Tasks on a Single Machine

Dependent and non-preemptive tasks

NP-hardness even with unit processing times for the problems $\mathbf{1|p_j=1,prec|\Sigma U_i}$ and $\mathbf{1|p_j=1,prec|\Sigma T_i}$.

Proof. ***Clique problem***: for a given graph $G(V,E)$ and a positive integer k determine if G contains a complete subgraph of k vertices.

CP ➔ $\mathbf{1|p_j=1,prec|\Sigma U_i}$ reduction: We put unit processing time tasks T_v with $d_v=|V\cup E|$ for every vertex $v\in V$ and T_e with $d_e=k+k(k-1)/2$ for every edge $e\in E$. Precedence constraints: $T_v \prec T_e \Leftrightarrow v$ is incident to e. Limit $L=|E|-k(k-1)/2$.

All the tasks are completed until $|V\cup E|$ in any optimal solution. Thus, whenever $\Sigma U_i \le L$, at least $k(k-1)/2$ tasks T_e are completed until $k+k(k-1)/2$. Therefore the corresponding edges are incident with at least k vertices (for which the tasks T_v precede T_e). That is possible only in the case these k vertices form a clique.

In a similar way CP ➔ $\mathbf{1|p_j=1,prec|\Sigma T_i}$ reduction.

Minimizing the Number of Tardy Tasks on a Single Machine

Parallel processors, minimizing $C_{\max}$ again

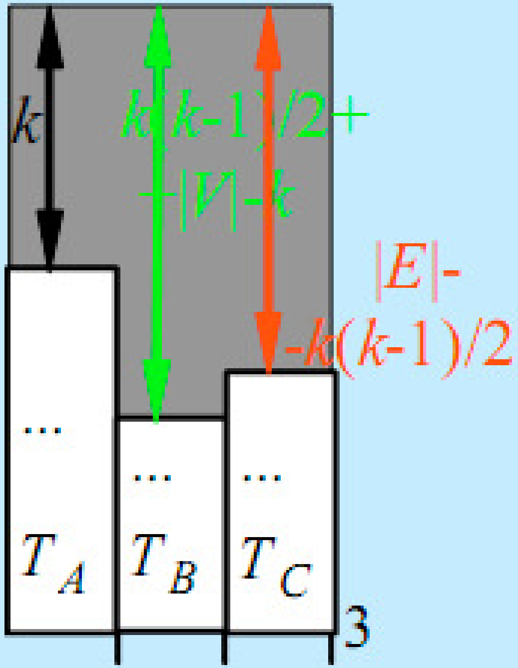
We have constructed a reduction PK ➔ $\mathbf{1|p_j=1,prec|\Sigma U_i}$. In a similar way NP–hardness of $P|p_j=1,prec|C_{\max}$ can be proved.

Proof. *Clique Problem*: for a given graph $G\ (V,E)$ and a positive integer k determine if G contains a complete subgraph of k vertices.

CP ➔ $P|p_j=1,prec|C_{\max}$ reduction: unit time tasks T_v for each $v \in V$ and T_e for each $e \in E$. Precedence constraints: $T_v \prec T_e \Leftrightarrow v$ is incident to e. Limit $L=3$. Moreover, 3 'levels' of unit tasks $T_{A1}, T_{A2}, \ldots \prec T_{B1}, T_{B2}, \ldots \prec T_{C1}, T_{C2}, \ldots$ The number of processors m big enough to make $C_{\max}=3$ possible:

If $C_{\max}^*=3$ then:

- All the gray-colored parts are fulfilled with T_v and T_e,
- In the 1st time unit only T_v are scheduled, in the 3rd time unit only T_e are scheduled,
- In the 2nd time unit $k(k-1)/2$ tasks T_e are processed and the corresponding edges are incident to k vertices, which form a clique (corresponding tasks are processed in the first time unit).



Scheduling on Dedicated Processors

Remainder

- jobs consist of operations preassigned to processors (T_{jj} is an operation of J_j that is preassigned to M_i, its processing time is p_{ij}). A job is completed when the last its operation is completed,
- some jobs may not need all the processors (*empty operations*),
- no two operations of the same job can be scheduled in parallel,
- processors are capable to process at most one operation at a time.

Models of scheduling on dedicated processors:

- ***flow shop*** – all jobs have the same processing order through the machines coincident with machines numbering,
- ***open shop*** – no predefined machine sequence exists for any job,
- other, not discussed ...

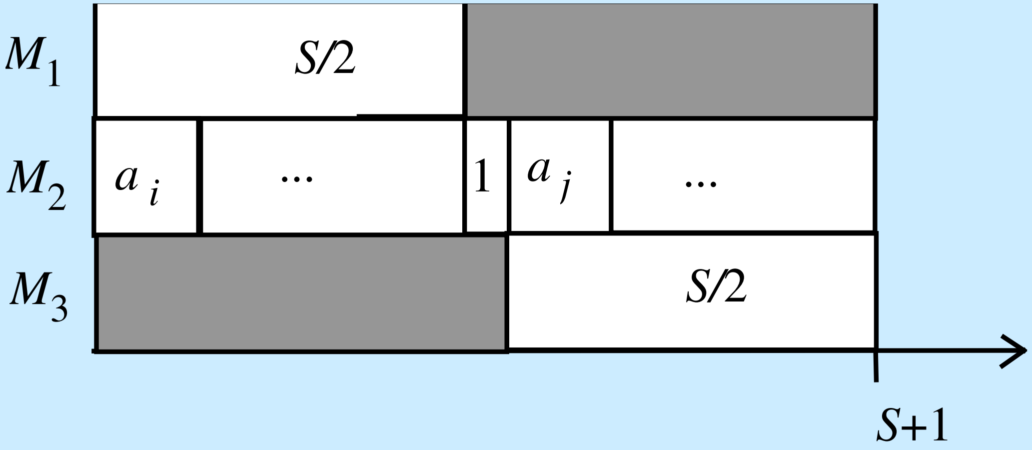
Scheduling on Dedicated Processors

Flow shop

Even 3-processor scheduling ($F3||C_{\max}$) is NP-hard.

Proof. ***Partition problem***: for a given sequence of positive integers $a_1,\ldots a_n$, $S=\Sigma_{i=1,\ldots,n}\, a_i$ determine if there exists a sub-sequence of sum $S/2$?

$PP \rightarrow F3||C_{\max}$ reduction: put n jobs with processing times $(0,a_i,0)$ $i=1,\ldots,n$ and a job $(S/2,1,S/2)$. Determine of these jobs can be scheduled with $C_{\max} \leq S+1$.



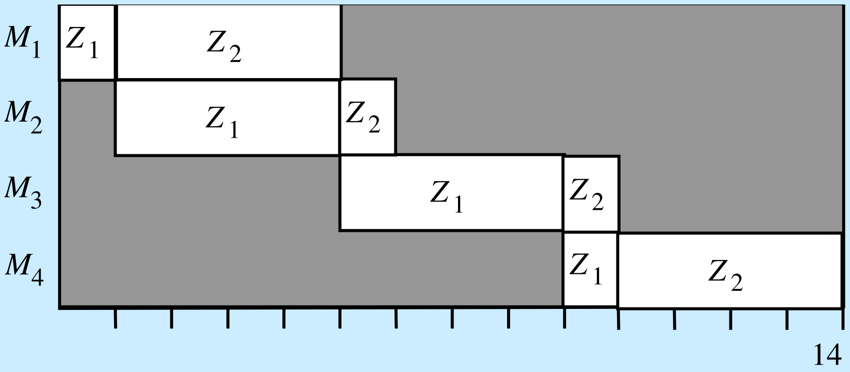
Permutation Flow Shop (***PF***): flow shop + each machine processes jobs in the same order (some permutation of the jobs).

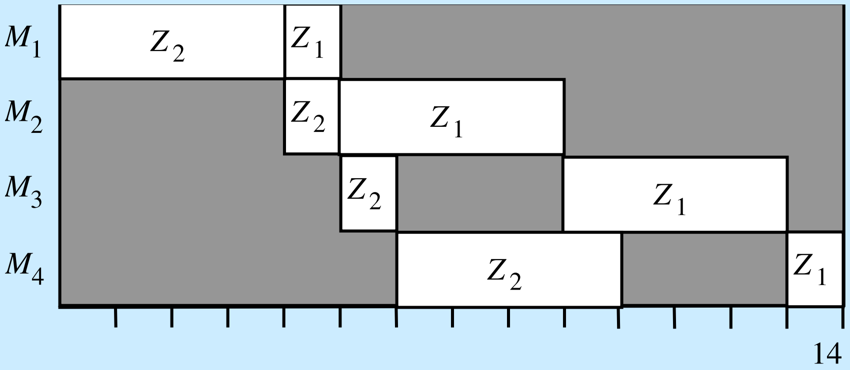
Scheduling on Dedicated Processors

Flow shop

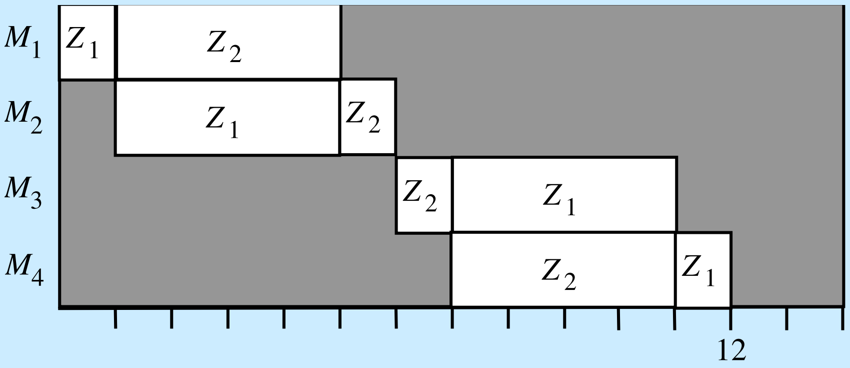
In a classical flow shop the jobs visit the processors in the same order (coincident with processors numbers), however the sequeneces of jobs within processors may differ (which may occur even in an optimal schedule).

Example. m=4, n=2. Processing times (1,4,4,1) for Z_1 and (4,1,1,4) for Z_2.





Permutation schedules ...



and non-permutation schedule

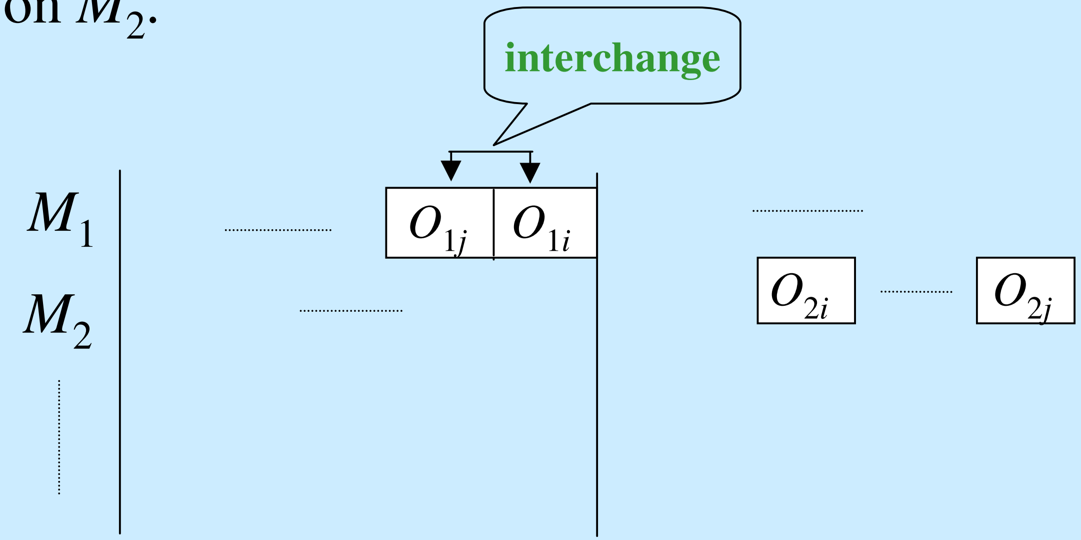
Scheduling on Dedicated Processors

Flow shop

Suppose $p_{ij}>0$. There exists an optimal schedule, such that the sequence of jobs for the first two processors is the same and for the last two processors is the same.

Corollary. An optimum schedule **$PFm||C_{\max}$** is an optimum solution **$Fm||C_{\max}$** for $m\leq3$ and $p_{ij}>0$ (only permutation schedules are to be checked – smaller space of solutions to search!).

Proof. The sequence of jobs on M_1 can be rearranged to be coincident with the sequence on M_2.



Scheduling on Dedicated Processors

Flow shop

Two processors scheduling $F2||C_{\max}$ (includes the case of preemptive scheduling ($F2|pmtn|C_{\max}$), ***Johnson algorithm*** $\boldsymbol{O(n \log n)}$**:**

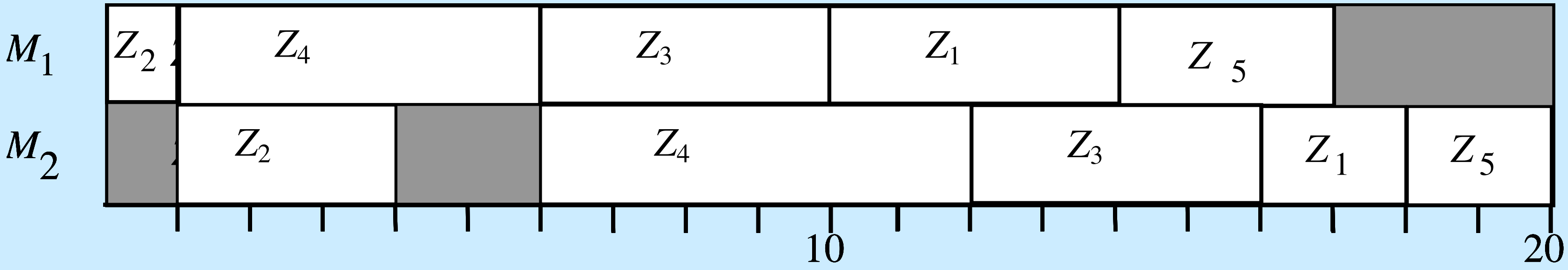
1. Partition the set of jobs into two subsets $N_1=\{Z_j\colon p_{1j}<p_{2j}\}$, $N_2=\{Z_j\colon p_{1j}\geq p_{2j}\}$,
2. Sort N_1 in non-decreasing p_{1j} order and N_2 in non-increasing p_{2j} order,
3. Schedule all the jobs on both machines in order of the concatenation sequence N_1,N_2.

Example. Johnson algorithm, m=2, n=5.

	Z_1	Z_2	Z_3	Z_4	Z_5
M_1	4	$\boxed{1}$	4	$\boxed{5}$	3
M_2	$\boxed{2}$	3	$\boxed{4}$	6	$\boxed{2}$
	N_2	N_1	N_2	N_1	N_2

N_1:	Z_2	Z_4
	1	5

N_2:	Z_3	Z_1	Z_5
	4	2	2



Scheduling on Dedicated Processors

Flow shop

- $F2||\Sigma C_j$ is NP-hard,
- $F3||C_{\max}$, in the case M_2 is ***dominated*** by M_1 ($\forall_{i,j}\ p_{1i} \geq p_{2j}$) or by M_3 ($\forall_{i,j}\ p_{3i} \geq p_{2j}$) one can use Johnson algorithm for n jobs with processing times ($p_{1i}+p_{2i}$, $p_{2i}+p_{3i}$), $i=1,\ldots,n$.

$F||C_{\max}$: polynomial-time „graphical" algorithm for $n=2$ jobs and arbitrary number of machines. Sketch:

1. We put intervals of the length p_{11}, p_{21}, ..., p_{m1} (processing times of J_1) on the OX axis and we put invervals of the length p_{12}, p_{22}, ..., p_{m2} on the OY axis.
2. We create rectangular obstacles – Cartesian products of corresponding intervals (a processor cannot process two tasks at a time).
3. We construct the shortest path consisting of segments parallel to one of the axis (single processor is working) or diagonal in the plane (both processors are working), avoiding passing through any obstacles, from (0,0) to ($\Sigma_i p_{i1}$, $\Sigma_i p_{i2}$) – the distance function is defined by $d((x_1,x_2),(y_1,y_2)) = \max\{|x_1-x_2|,|y_1-y_2|\}$. The length of the path is qeual to the length of the optimim schedule.

Scheduling on Dedicated Processors

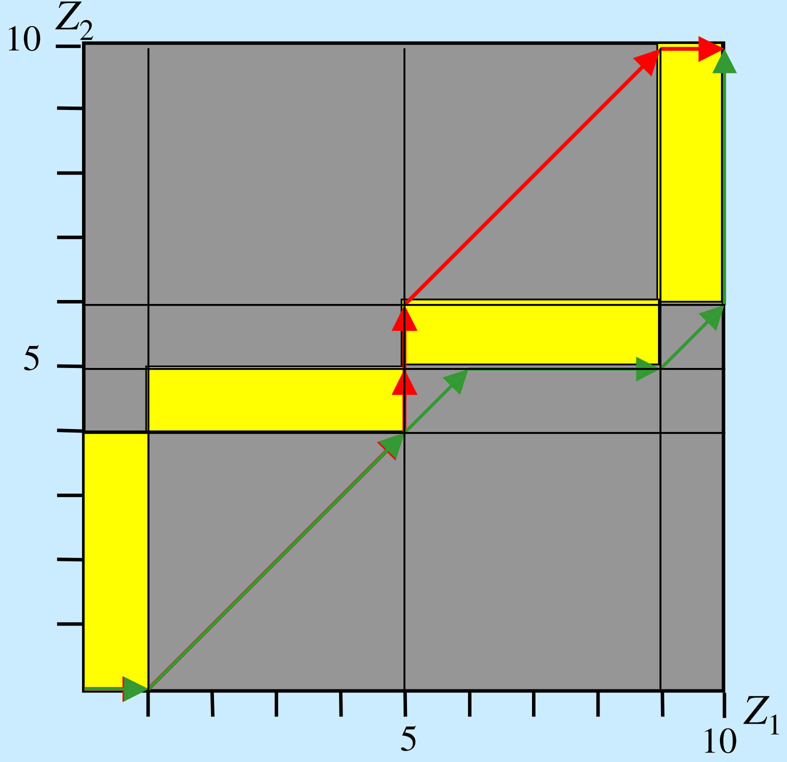
Flow shop

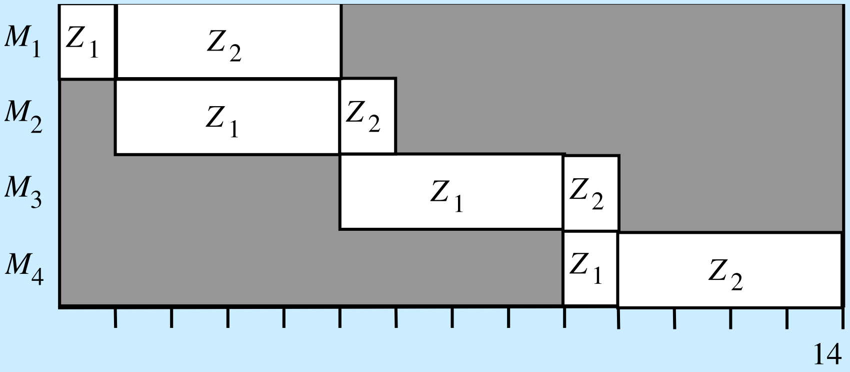
Example. Graphical algorithm.

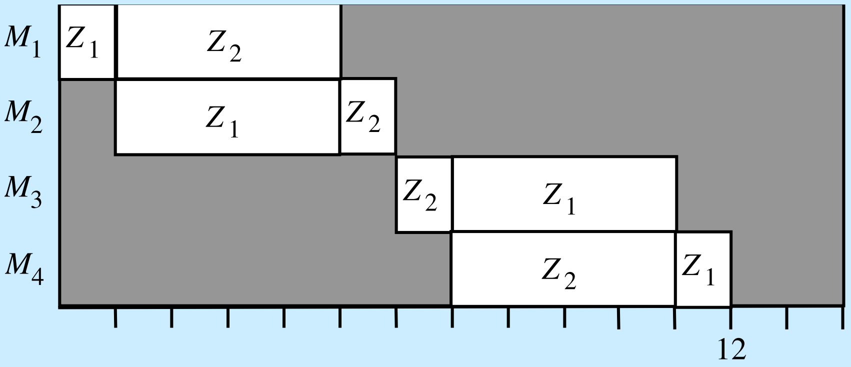
m=4, n=2 and

Z_1 processing times (1,4,4,1);

Z_2 processing times (4,1,1,4).





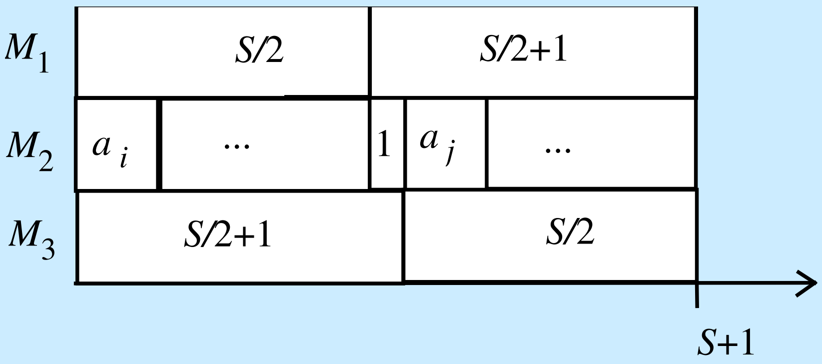


Scheduling on Dedicated Processors

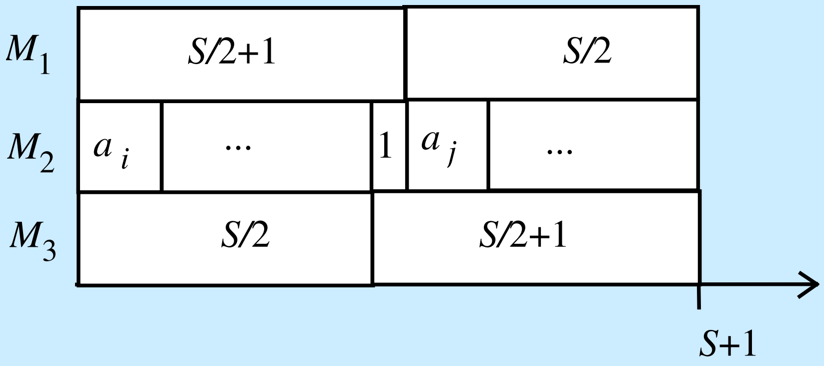
Open shop

The three processors case ($O3||C_{\max}$) is NP-hard.

Proof. $PP \rightarrow O3||C_{\max}$ reduction: put n tasks with processing times $(0,a_i,0)$ $i=1,\ldots,n$ and three tasks of processing times $(S/2,1,S/2)$, $(S/2+1,0,0)$, $(0,0,S/2+1)$. Determine if there exisits a schedule with $C_{\max} \le S+1$.



or



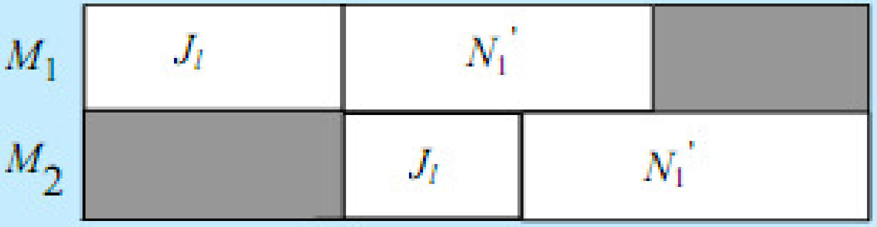
- The problem $O2||\Sigma C_j$ is NP-hard.

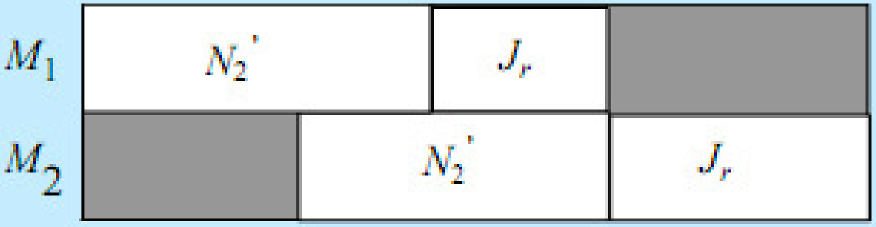
Scheduling on Dedicated Processors

Open shop

The case of two processors $O2||C_{\max}$ (and $O2|pmtn|C_{\max}$), *Gonzalez–Sahni algorithm* $O(n)$:

1. Partition the set of tasks into $N_1 = \{J_j : p_{1j} < p_{2j}\}$, $N_2 = \{J_j : p_{1j} \geq p_{2j}\}$,
2. Let J_r, J_l, be two jobs such that: $p_{1r} \geq \max_{Jj \in N2} p_{2j}$; $p_{2l} \geq \max_{Jj \in N1} p_{1j}$;
3. $p_1 := \Sigma_i p_{1i}$; $p_2 := \Sigma_i p_{2i}$; $N_1' := N_1 \setminus \{J_r, J_l\}$; $N_2' := N_2 \setminus \{J_r, J_l\}$;
4. Construct two schedules: jobs $N_1' \cup \{J_l\}$, Z_l as the first; $N_2' \cup \{J_r\}$, J_r as the last (permutation and no–idle schedules) :

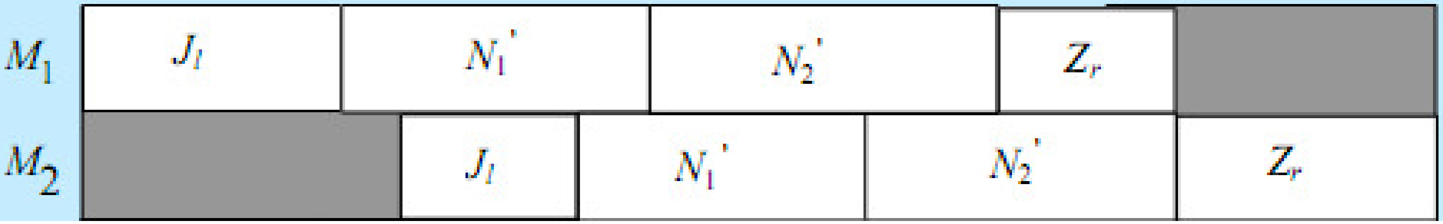




5. Merge these schedules. **if** $p_1 - p_{1l} \geq p_2 - p_{2r}$ ($p_1 - p_{1l} < p_2 - p_{2r}$)

   then 'shift' the tasks of $N_1' \cup \{J_l\}$ to the right on M_2

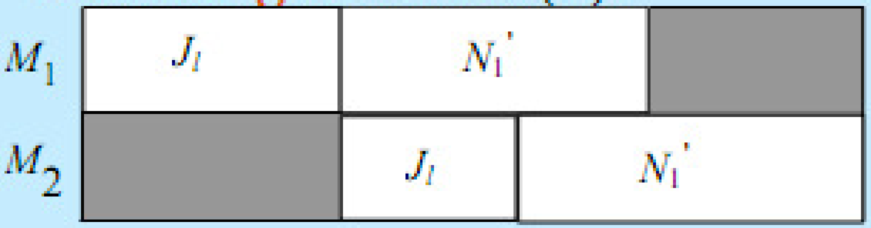
   else 'shift' the tasks of $N_2' \cup \{J_r\}$ to the left on M_1;

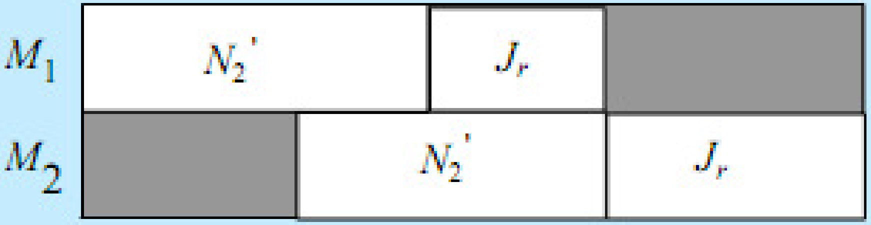


Scheduling on Dedicated Processors

Open shop

The case of two processors $O2||C_{\max}$ (and $O2|pmtn|C_{\max}$), *Gonzalez–Sahni algorithm* **$O(n)$:**

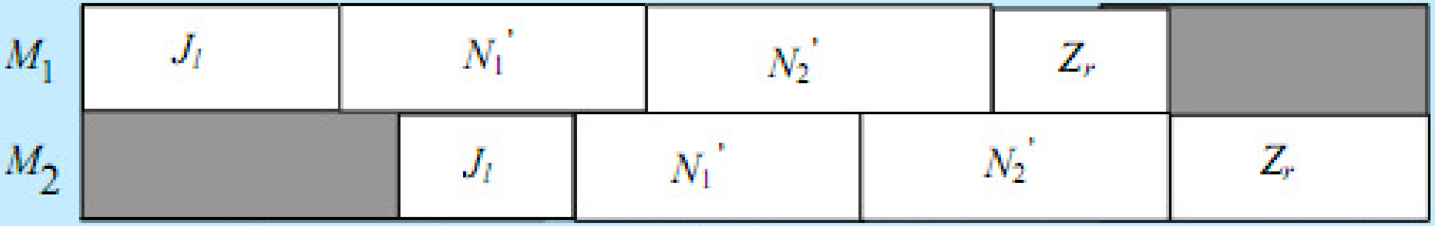




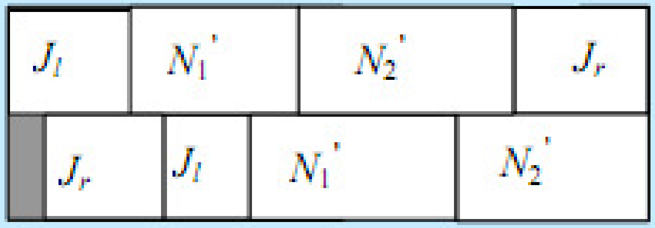
5. Merge these schedules. **if** $p_1-p_{1l} \geq p_2-p_{2r}$ ($p_1-p_{1l}<p_2-p_{2r}$)

then 'shift' the tasks of $N_1' \cup \{J_l\}$ to the right on M_2

else 'shift' the tasks of $N_2' \cup \{J_r\}$ to the right on M_1; [*]

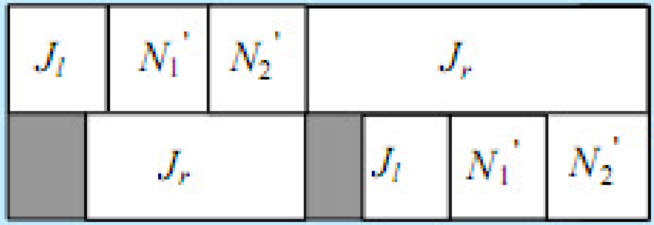


6. Move the operation of J_r on M_2 ([*] J_l on M_1) to the beginning ([*] the end) and then to the right ([*] to the left).



$C_{\max}=$
$=\max\{p_1,p_2\}$

or



$C_{\max}=$
$=\max\{p_1,$
$p_{1r}+p_{2r}\}$

Scheduling on Dedicated Processors

Open shop

Example. Gonzalez–Sahni algorithm, m=2, n=5.

	J_1	J_2	J_3	J_4	J_5
M_1	4	$\boxed{1}$	4	$\boxed{5}$	3
M_2	$\boxed{2}$	3	$\boxed{4}$	6	$\boxed{2}$
	N_2	**N_1**	**N_2**	**N_1**	**N_2**

N_1: J_2, J_4

N_2: J_1, J_3, J_5

$J_r := J_1$

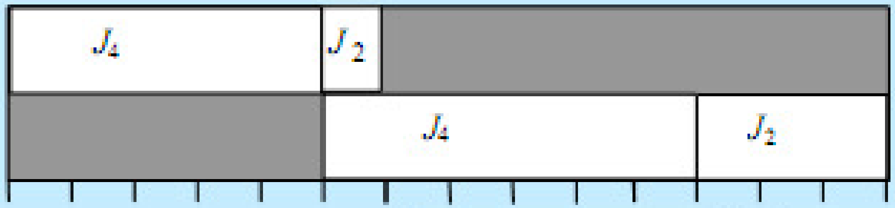
$J_l := J_4$

$p_{1r} \geq \max_{Jj \in N2} p_{2j} = 4$

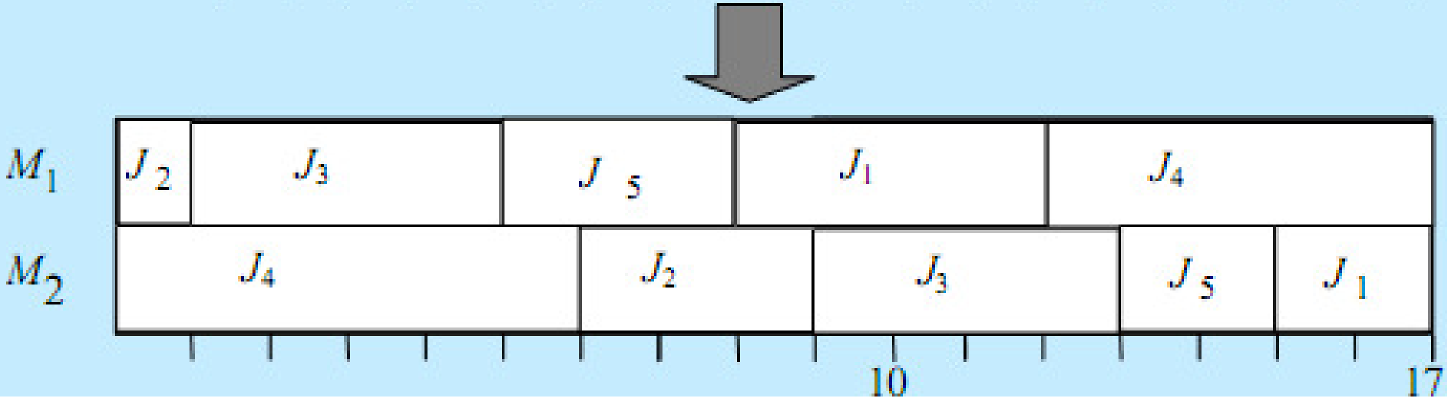
$p_{2l} \geq \max_{Jj \in N1} p_{1j} = 5$

N_1': J_2

N_2': J_3, J_5



Merge:

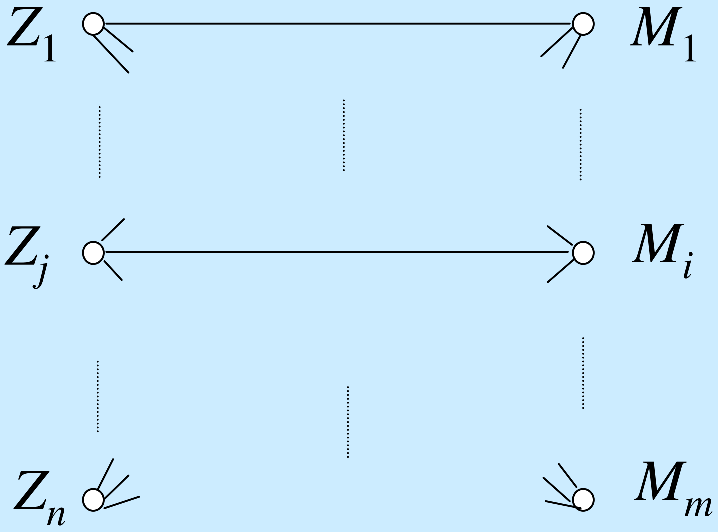


Scheduling on Dedicated Processors

Open shop

Zero or unit processint times (***$O|ZUET|C_{\max}$***): ***polynomial-time algorithm*** based on edge-coloring of bipartite graphs.

1. Bipartite graph *G*:
   a) one partition correspond to the job set; the other represents the processors,
   b) each non-empty operation O_{ij} corresponds to an edge $\{Z_j, M_i\}$.



2. We edge-color this graph using $\Delta(G)$ colors. The colors are interpreted as the time units in which the corresponding tasks are scheduled, (**proposal:** feasible schedule ⇔ proper coloring).

3. Then $C_{\max}^{*} = \Delta(G) = \max\{\max_i \Sigma_{j=1,\dots,n} p_{ij}, \max_j \Sigma_{i=1,\dots,m} p_{ij}\}$. Obviously, no shorter schedule exists.

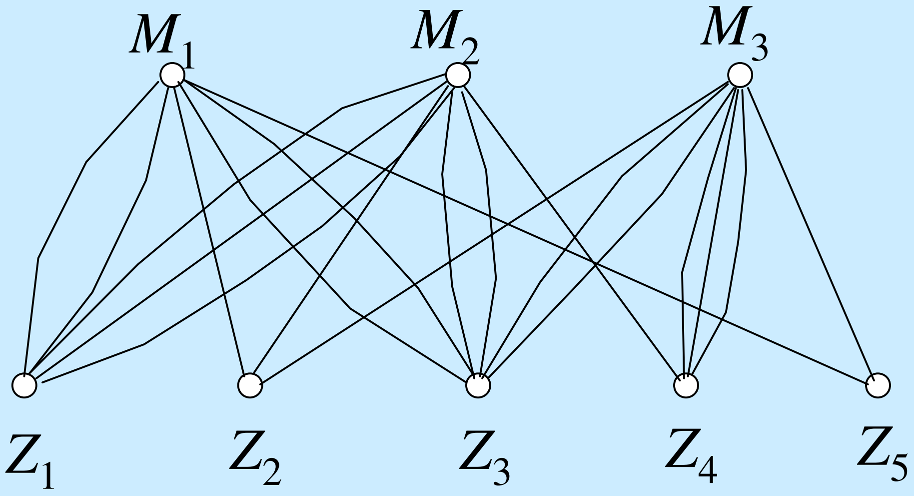
Scheduling on Dedicated Processors

Open shop

Preemptive scheduling ($O|pmtn|C_{\max}$): ***pseudo-polynomial algorithm*** similar to the algorithm for $O|ZUET|C_{\max}$. We construct a bipartite multigraph G, i.e. each non-empty task T_{ij} corresponds to p_{ij} parallel edges. Again $C_{\max}{}^* = \max\{\max_i \Sigma_{j=1,\ldots,n} p_{ij}, \max_j \Sigma_{i=1,\ldots,m} p_{ij}\}$.

Why „pseudo"? The number of edges may be non-polynomial ($=\Sigma_{i=1,\ldots,m;\ j=1,\ldots,n}\ p_{ij}$), the schedule may contain non-polynomial number of interrupts.

Example. Preemptive scheduling m=3, n=5, p_1=(2,3,0), p_2=(1,1,1), p_3=(2,2,2), p_4=(0,1,3), p_5=(1,0,1).



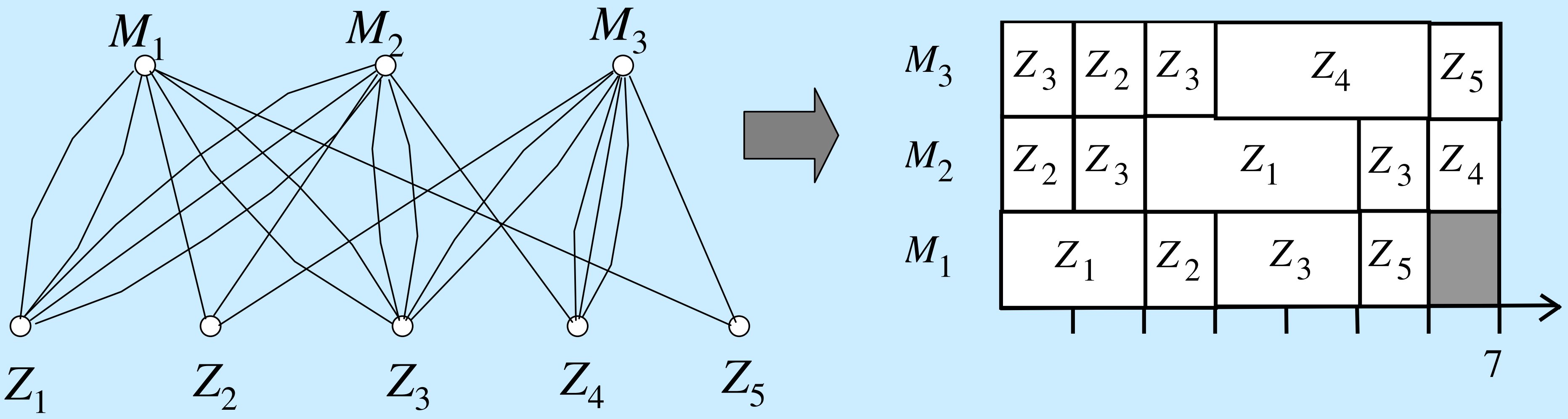
Scheduling on Dedicated Processors

Open shop

<u>Preemptive scheduling</u> ($O|pmtn|C_{\max}$): ***pseudo-polynomial algorithm*** similar to the algorithm for $O|ZUET|C_{\max}$. We construct a bipartite multigraph G, i.e. each non-empty task T_{ij} corresponds to p_{ij} parallel edges. Again $C_{\max}{}^{*} = \max\{\max_i \Sigma_{j=1,\dots,n} p_{ij}, \max_j \Sigma_{i=1,\dots,m} p_{ij}\}$.

Why „pseudo”? The number of edges may be non-polynomial ($=\Sigma_{i=1,\dots,m;\, j=1,\dots,n}\, p_{ij}$), the schedule may contain non-polynomial number of interrupts.

Example. Preemptive scheduling $m=3$, $n=5$, $p_1=(2,3,0)$, $p_2=(1,1,1)$, $p_3=(2,2,2)$, $p_4=(0,1,3)$, $p_5=(1,0,1)$.



Scheduling on Dedicated Processors

Open shop

Preempttive scheduling ($O|pmtn|C_{\max}$):

- ***polynomial time algorithm*** is known; it is based on *fractional edge-coloring* of weighted graph (each task T_{ij} corresponds to an edge $\{J_j, M_i\}$ of weight p_{ij} in graph G),

Minimizing $C_{\max}$ on parallel processors ... again

Polynomial-time algorithm for $R|pmtn|C_{\max}$.

$R|pmtn|C_{\max} \rightarrow O|pmtn|C_{\max}$ reduction: Let x_{ij} be the part of T_j processed by M_i (in the time $t_{ij}= p_{ij}x_{ij}$). If we knew optimal values x_{ij}, we could use the above algorithm treating these tasks' parts as tasks of preemptive jobs (constraints for the schedule are the same!).

How to derive these values? We transform the scheduling problem to linear programming:

minimize C such that:

$$\Sigma_{i=1,\ldots,m}\, x_{ij}=1,\ j=1,\ldots,n$$

$$C \geq \Sigma_{j=1,\ldots,n}\, p_{ij}x_{ij},\ \ i=1,\ldots,m, \qquad C \geq \Sigma_{i=1,\ldots,m}\, p_{ij}x_{ij},\ j=1,\ldots,n.$$