CICESE Research Center
Ensenada, Baja California
Mexico

Introducción a Calendarización en Sistemas Paralelas, Grids y Nubes

Dr. Andrei Tchernykh
CICESE
Centro de Investigación Científica y de Educación Superior de Ensenada
Ensenada, Baja California, México, chernykh@cicese.mx
http://usuario.cicese.mx/~chernykh/

Escuela de Modelación y Métodos Numéricos
CIMAT, Guanajuato, México
25-28 de junio de 2014

The first problem

- set of independent tasks
- identical processors
- minimize schedule length.

Complexity:

- the problem is not easy to solve since even simple cases such as scheduling on two processors can be proved to be NP-hard.

There is no hope of finding a polynomial time algorithm

Solution:

- find an approximation algorithm for the original problem
- evaluate its worst case
- mean behavior.

One of the most often used general approximation strategies for solving scheduling problems is *list scheduling*

- priority list of the tasks is given
- at each step the first available processor is selected to process the first available task on the list

The accuracy of a given list scheduling algorithm depends on the order in which tasks appear on the list.

Set of tasks $\mathcal{T} = \{T_1, T_2, ..., T_n\}$

Set of processors $\mathcal{P} = \{P_1, P_2, ..., P_m\}$

p_j = *processing time of task* T_j

Identical Processors. List Scheduling

$W_{seq} = \sum_{i=1}^{n} p_i$ be the total work of all jobs

$p_{\max}$ is the maximum processing time of a job.

W_{idle} be the total idle intervals, $W_{idle} \le p_{\max}(m-1)$

$C \le \frac{W_{seq} + W_{idle}}{m}$ is the completion time of the set of tasks.

$$C \le \frac{W_{seq} + p_{\max}(m-1)}{m} \le \frac{W_{seq} - p_{\max}}{m} + p_{\max}, \quad C \le \frac{W_{seq}}{m} + \frac{(m-1)}{m} p_{\max}$$

$\frac{W_{seq}}{m}$ and $p_{\max}$ are lower bounds of C_{opt}^{seq}, it follows that the worst-case performance bound is $\rho^{seq} \le 2 - \frac{1}{m}$.

Outline

Preliminaries

1.1 Scheduling in Processor and Operating Systems
1.2 Production Scheduling
1.3 Control Systems
1.4 Basic Notions
1.5 The Scheduling Model
- Deterministic Model
- Optimization Criteria
- Scheduling Problem Notation
- Scheduling Algorithms

1.6 *Schedule representation* and evaluation
1.7 Three-field notation

Outline

Scheduling on Parallel Processors

2. Minimizing Schedule Length
   - Identical Processors
     - Independent tasks
     - List Scheduling
     - *LPT* Algorithm
     - Preemptions. *McNaughton's rule*
     - Precedence constraints
     - Graham anomalies
3. Minimizing Due Date Involving Criteria
   - Identical Processors
     - EDF - Earliest Deadline First scheduling
     - LL-Least Laxity scheduling
     - Minimizing number of tardy tasks

Outline

4. Communication Delays
   - Scheduling without Task Duplication
   - Scheduling with Task Duplication
5. Bin Packing Problem
   - Next Fit and First Fit
   - *Best Fit* (BF) and *Worst Fit* (WF).
   - One-Dimensional Bin Packing Problem
   - Two-Dimensional Bin Packing Problem
6. Strip Packing Problem
   - Bottom-left algorithm
   - Level-oriented algorithms
   - Split algorithms
   - Shelf algorithms
   - Hybrid algorithms
   - The Slave Algorithm
   - On-line v's Off-line

Outline

References

1. J. Blazewicz, K. Ecker, G. Schmidt, J. Weglarz, Scheduling in Computer and Manufacturing Systems, Springer, pp. 495, 2001 ISBN:3540419314
2. Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Edited by Joseph Y-T. Leung. Published by CRC Press, Boca Raton, FL, USA, 2004
3. Handbook on Scheduling. From Theory to Applications, by J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, J. Weglarz, Springer, pp. 647, 2007 ISBN:978-3-540-28046-0

Topic 1: Preliminaries

Objective
Application areas
Basic Notions
The Scheduling Model

Introduction

Application Area:

- Scheduling in Processor and Operating Systems
- Production Scheduling
- Technical and Industrial Processes
- Control Systems

Application Area: *Scheduling in Processor and Operating Systems*

In *operating systems* :

- hundreds of processes waiting to get access to the processor
- strategy takes various parameters into account, such as
    - priority of the process
    - its parent priority
    - already consumed CPU time
    - assigned resources

The scheduler (process dispatcher) is designed to optimize some system performance:

- optimizing throughput: maximize the number of completed processes per time unit
- minimizing the makespan
- maximizing profit for the owner of the machine

Application Area: *Scheduling in Processor and Operating Systems*

Questions regarding the scheduling of activities in computers occur at different levels:

- *Inside processors* : sequencing of micro-operations; pipelining
- scheduling strategies in *single processor operating systems* :
  - round robin
  - priority based dispatcher algorithms
- *multiprocessor systems*, consisting of a CPU, co-processors, and I/O processors:
  - process handling,
  - assignment of activities to the special purpose processors
- *parallel processing* on a large number of identical processors as in massive parallelism:
  - Work distribution, taking into account the network connectivity and communication delays

Application Area: *Scheduling in Processor and Operating Systems*

- *distributed processing* (several computers (workstations, PC's, etc.) are connected in a local area network (LAN), metropolitan area network (MAN), wide area network (WAN), Grids, Clouds
- *real-time operating systems* in parallel or distributed systems need careful handling of activities with deadlines

Another example of practical interest concerns *production systems*

Typical in this area is the demand for optimal working plans for assembly lines and for flexible manufacturing machines, e.g. in production cells

General requirements:

- production due dates
- resource balancing
- maximal production throughput
- minimum storage cost

Application Area: *Production Scheduling*

Examples:

- Control of robot movement has to deal with data, and concerns the real time coordination of moving the arm(s)
- Assembly lines are of pipeline structure; their optimal design leads to flow shop problems
- Organizing flexible manufacturing machines leads to problems of optimizing lot sizes under the requirement of optimal throughput while minimizing overhead due to tool change delays and other setup costs
- Optimal routing of automated guided vehicles (AGV's) leads to questions that again require careful planning and sequencing

In a manufacturing environment deterministic scheduling is also known as *predictive*

Its complement is *reactive scheduling*, which can also be regarded as deterministic scheduling with a shorter planning horizon

Application Area: *Technical and Industrial Processes*

Activities from

- production planning
- computer aided design
- work planning
- manufacturing
- quality control

have to be coordinated

The objectives are similar:

- optimal capacity planning
- maximal throughput
- minimum storage cost, etc.

Application Area: *Control Systems*

In *real-time systems* the particular situation dictates conditions different from those before:

> some processes must be activated *periodically* with a fixed rate, and others have to meet given *deadlines*

In such systems, meeting the deadlines can be a crucial condition for the correct operation of the environment

Examples of application areas are

- aircraft control
- power plants, heat control, turbine speed control,
- frequency and voltage stabilization etc.,

Basic Notions

Basic Notions. Introduction

The notion of *task* is used to express some well-defined activity or piece of work

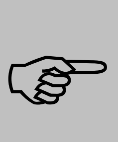
Planning in practical applications requires some knowledge about the tasks

This knowledge does not regard their nature, but rather general properties such as

- processing times,
- relations between the tasks concerning the order in which the tasks can be processed,
- release times which inform about the earliest times the tasks can be started,
- deadlines that define the times by which the tasks must be completed,
- due dates by which the tasks should be completed together with cost functions that define penalties in case of due date violations,
- additional resources (for example, tools, storage space, data)

Based on these data one could try to develop a *work plan* or *time schedule* that specifies for each task when it should be processed, on which machine or processor, including preemption points, etc.

Depending of how much is known about the tasks to be processed, we distinguish between three main directions in scheduling theory:

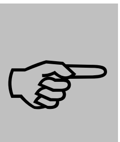


Deterministic or *static* or *off-line scheduling* assumes that all information required to develop a schedule is known in advance, before the actual processing takes place

Especially in production scheduling and in real-time applications the deterministic scheduling discipline plays an important role

⇒ Stand-alone theory, known as *deterministic scheduling theory*

Characteristically, one has to deal with the development of optimal algorithms



Non-deterministic scheduling is less restrictive: only partial information is known

for example computer applications where tasks are pieces of software with unknown run-time

Basic Notions. Introduction

On-line scheduling: In many situations detailed knowledge of the nature of the tasks is available, but the time at which tasks occur is open

If the demand of executing a task arises a decision upon acceptance or rejection is required, and, in case of acceptance, the task start time has to be fixed

In this situation schedules cannot be determined off-line, and we then talk about *on-line* scheduling or *dynamic* scheduling

Non-clairvoyant scheduling: consider problems of scheduling jobs with unspecied execution time requirements

Stochastic scheduling: only probabilistic information about parameters is available

In this situation probability analysis is typical means to receive information about the system behavior

- For each type of scheduling one can find justifying applications

Here, off-line scheduling (occasionally also on-line scheduling) is considered

Deterministic Scheduling Problems

The deterministic scheduling or planning problems arising in different applications have often strong similarities

hence essentially the same basic model can be used

Common aspects in these applications:

- processes consist of complex activities to be scheduled
- modeled by means of tasks or jobs

- Tasks usually need one of the available *machines*, maybe even a special machine, and additional *resources* of limited availability
- Between tasks there are relations describing the relative order in which the tasks are to be performed

  order of task execution can be restricted by conditions like *precedence constraints*

- *Preemption* of task execution can be allowed or forbidden

Deterministic Scheduling Problems

- Timing conditions such as task *release times*, *deadlines* or *due dates* may be given

    In case of due dates *cost functions* may define *penalties* depending on the amount of lateness

- There may be conditions for *time lags* between pairs of tasks, such as setup delays
- In so-called *shop problems* sequences of tasks, each to be performed on some specified machine, are defined

    An example is the well-known flow shop or assembly line processing

Scheduling problems are characterized not only by the tasks and their specific properties, but also by information about the processing devices

Processors or *machines* for processing the tasks can be

- *identical*,
- can have different speeds (*uniform*), or
- their processing capabilities can be *unrelated*

Deterministic Scheduling Problems

The problem is to determine an appropriate *schedule*, i.e. one that satisfies all conditions imposed on the tasks and processors

A schedule essentially defines the start times of the tasks on a specified processor

Generally there may exist several possible schedules for a given set of tasks

An important condition describes the intended properties of a schedule, as defined by an *optimization criterion*

Common criteria are:

- minimization of the *makespan* of the total task set,
- minimization of the *mean waiting* time of the tasks

The optimization criterion allows to choose an appropriate schedule

Such schedules are then used as a planning basis for carrying out the various activities

Unfortunately, finding optimal schedules is in general a very difficult process

Except for simplest cases, these problems turn out to be NP-hard, and hence the time required computing an exact solution is beyond all practical means

In this situation, algorithmic approaches for *sub-optimal* schedules seem to be the only possibility

The Scheduling Model

- Deterministic Model
- Optimization Criteria
- Scheduling Problem and $\alpha \mid \beta \mid \gamma$ - Notation
- Scheduling Algorithms

The Scheduling Model. Deterministic Model

Tasks, Processors, etc.

Set of tasks $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$

Set of resource types $\mathcal{R} = \{R_1, R_2, \ldots, R_s\}$

Set of processors $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$

Examples of processors:

CPUs in e.g. a multiprocessor system

Computers in a distributed processing environment

Production machines in a production environment

Processors may be

- *parallel* : they are able to perform the same functions
- *dedicated* : they are specialized for the execution of certain tasks

The Scheduling Model. Deterministic Model

Parallel processors have the same execution capabilities

Three types of parallel processors are distinguished

- *identical* : if all processors from set $\mathcal{P}$ have equal task processing speeds
- *uniform* : if the processors differ in their speeds, but the *speed* b_i of each processor is constant and does not depend on the tasks in $\mathcal{T}$
- *unrelated* : if the speeds of the processors depend on the particular task unrelated processors are more specialized : on certain tasks, a processor may be faster than on others

The Scheduling Model. Deterministic Model

Characterization of a task T_j

– Vector of *processing times* $p_j = [p_{1j}, ..., p_{mj}]$, where p_{ij} is the time needed by processor P_i to process T_j

Identical processors : $p_{1j} = ... = p_{mj} = p_j$

Uniform processors : $p_{ij} = p_j / b_i, i = 1, ..., m$

p_j = *standard processing time* (usually measured on the slowest processor),

b_i is the *processing speed factor* of processor P_i

Processing times are usually not known a priori in computer systems

Instead of exact values of processing times one can take their estimate

However, in case of deadlines exact processing times or at least upper bounds are required

The Scheduling Model. Deterministic Model

Arrival time (or *release* or *ready time*) r_j … is the time at which task T_j is ready for processing

if the arrival times are the same for all tasks from $\mathcal{T}$, then $r_j = 0$ is assumed for all tasks

- *Due date* d_j … specifies a time limit by which T_j should be completed

  problems where tasks have due dates are often called "soft" real-time problems. Usually, penalty functions are defined in accordance with due dates

- *Penalty functions* G_j define penalties in case of due date violations

- *Deadline* $\tilde{d}_j$ … "hard" real time limit, by which T_j must be completed

- *Weight* (*priority*) w_j ... expresses the relative urgency of T_j

– *Preemption* / *non-preemption* :

A scheduling problem is called *preemptive* if each task may be preempted at any time and its processing is resumed later, perhaps on another processor

If preemption of tasks is not allowed the problem is called *non-preemptive*

Conditions among the set of tasks $\mathcal{T}$: precedence constraints

$T_i \prec T_j$ means that the processing of T_i must be completed before T_j can be started

We say that a *precedence relation* $\prec$ is defined on set $\mathcal{T}$
mathematically, a precedence relation is a *partial order*

The tasks in $\mathcal{T}$ are called *dependent*

if the *relation* $\prec$ is non-empty
otherwise, the tasks are called *independent*

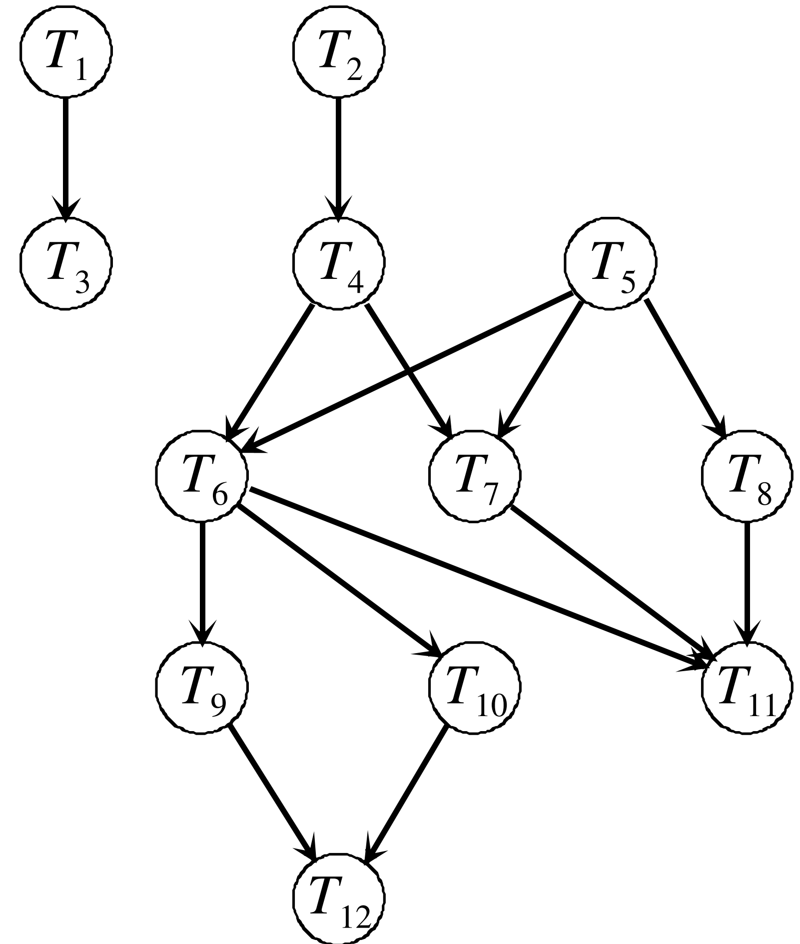
Representation of tasks with precedence constraints:

– *task-on-node graph (Hasse diagram)*

For each $T_i \prec T_j$, an edge is drawn between the corresponding nodes

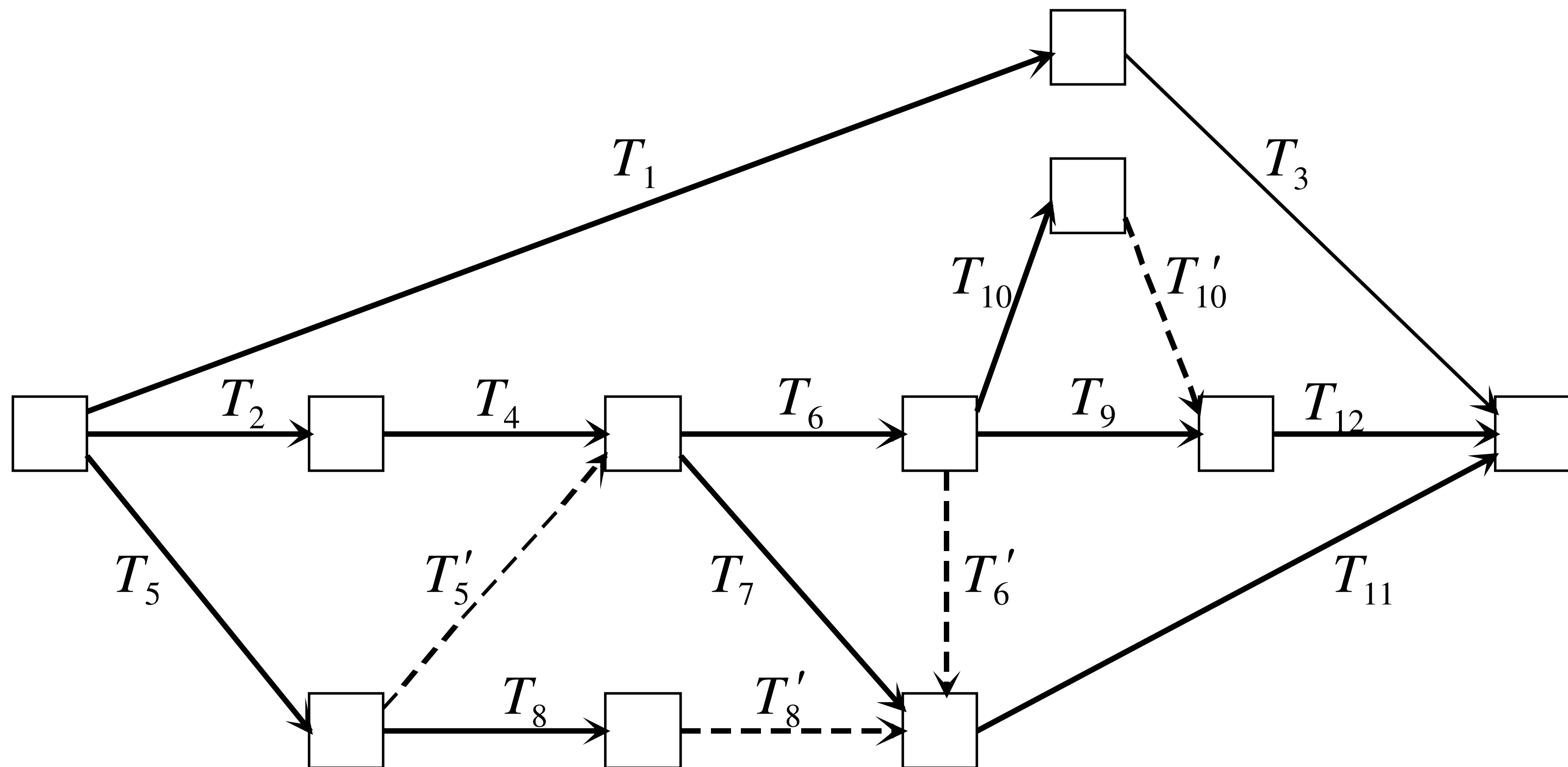
The situation $T_i \prec T_j$ and $T_j \prec T_k$ is called *transitive dependency* between T_i and T_k

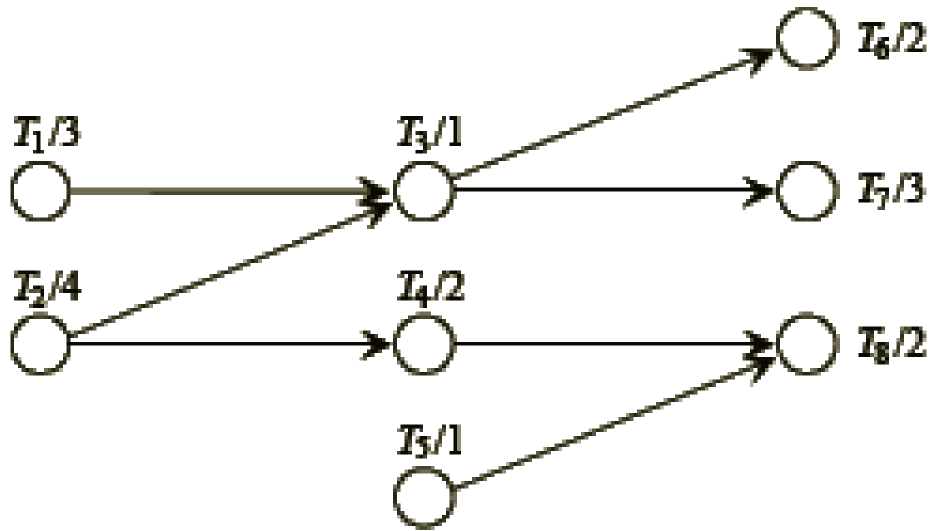
Transitive dependencies are not explicitly represented



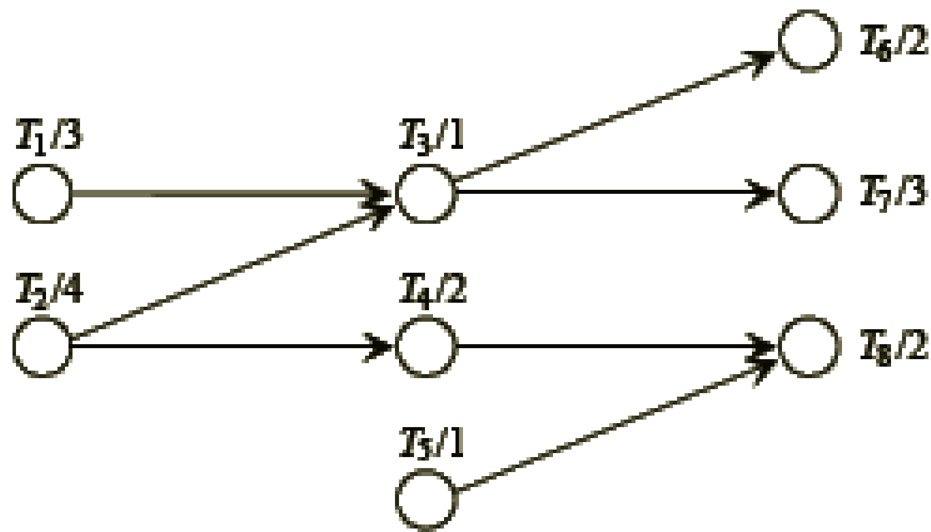
task-on-arc graph, *activity network.* Arcs represent tasks and nodes time events

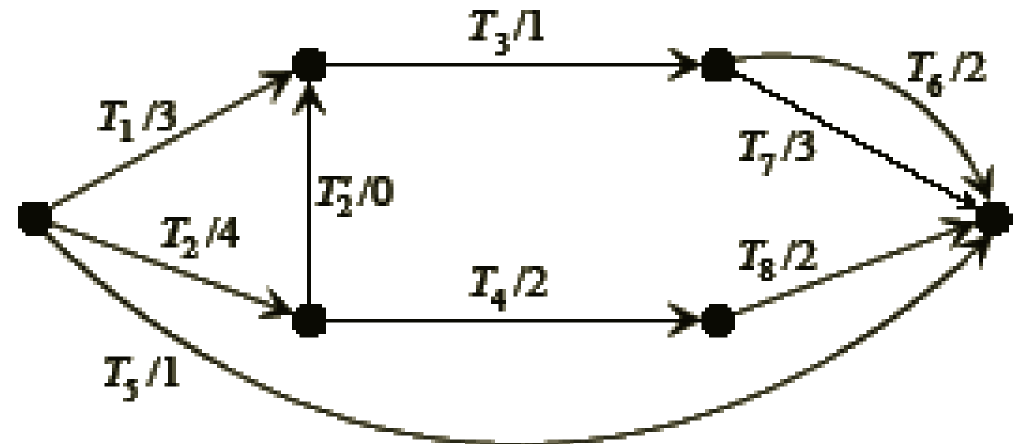
Example 1: $\mathcal{T} = \{T_1, ..., T_{10}\}$ with precedences as shown by the above Hasse diagram. A corresponding activity network:





task-on-arc graph ?????





Error????

Task T_j is called *available* at time t if $r_j \leq t$ and all its predecessors (with respect to the precedence constraints) have been completed by time t

Schedules

Schedules or work plans generally …

inform about the times and on which processors the tasks are executed

To demonstrate the principles, the schedules are described for the special case of:

- parallel processors
- tasks have no deadlines
- tasks require no additional resources

Release times and precedence constraints may occur

The Scheduling Model. Deterministic Model

A *schedule* $\mathcal{S}$ is an assignment of processors to the tasks from $\mathcal{T}$ (or an assignment of the tasks to the processors) such that:

– task T_j is processed in the time interval $[r_j, \infty)$ for p_j time units,

– all tasks are completed,

– at each instant of time, each processor works on at most one task,

– at each instant of time, each task is processed by at most one processor

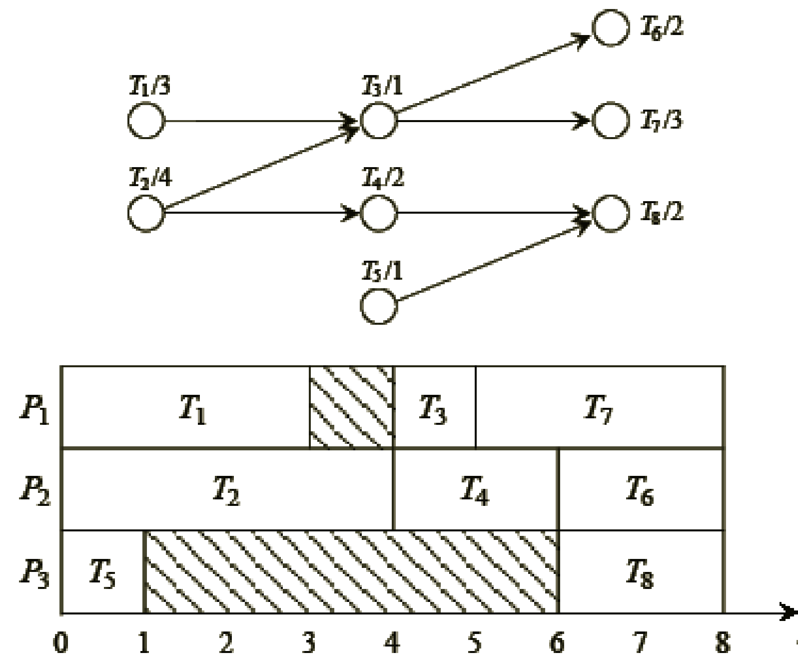
– if tasks T_i, T_j are in relation $T_i \prec T_j$ then the processing of T_j is not started before T_i has been completed,

– if T_j is no-preemptive then processing of T_j is not interrupted;
if T_j is preemptive then T_j may be interrupted only a *finite* number of times

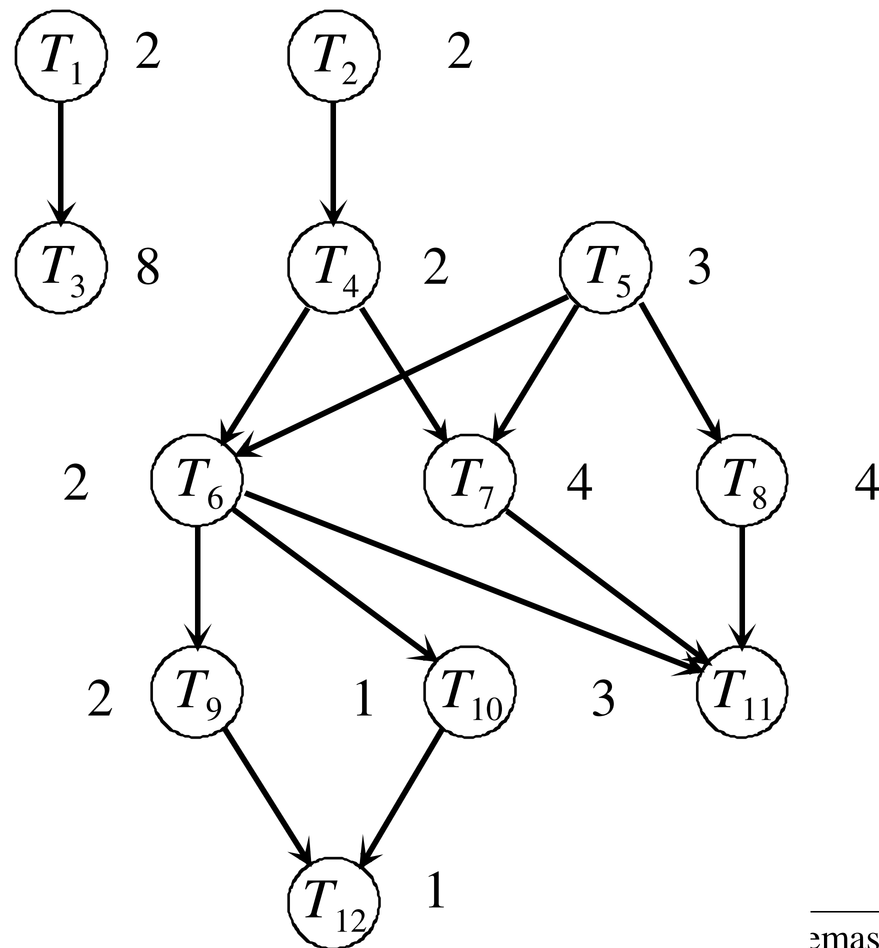
If all tasks are non-preemptive then the schedule is called *non-preemptive*

If all tasks are preemptive, then the schedule is called *preemptive*

(3) Graphic representation: Gantt chart - this is a two-dimensional diagram
The abscissa represents the time axis that usually starts with time 0 at the origin
Each processor is represented by a line
For a task T_j to be processed by P_i , a bar of length $p(T_j)$ and that begins at the time marked by $s(T_j)$, is entered in the line corresponding to P_i .



Example 1: $\mathcal{T} = \{T_1, ..., T_{12}\}$ with precedences as shown by the Hasse diagram:

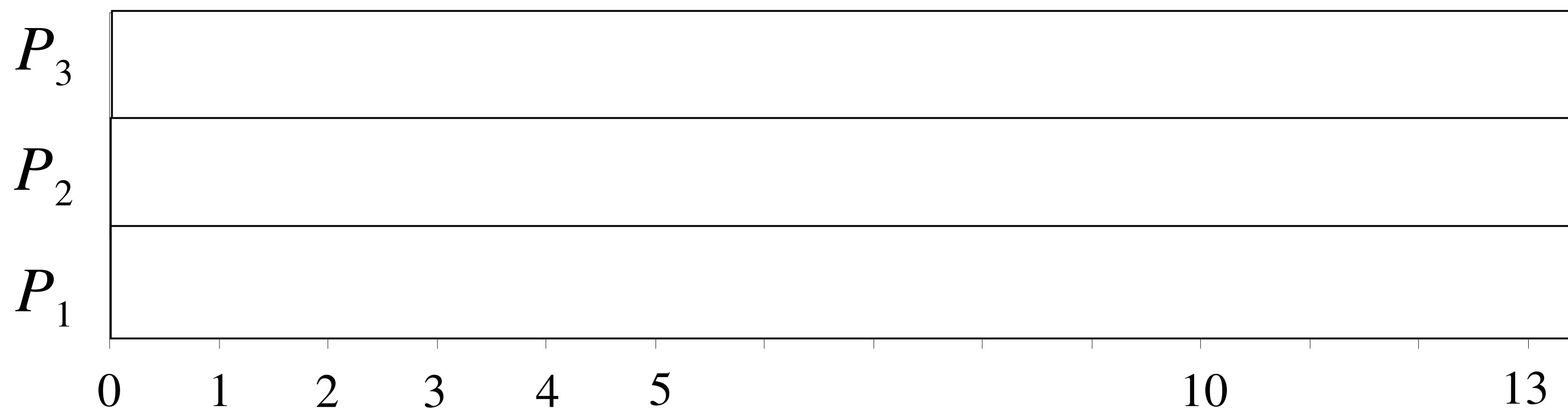


Example 2: non-preemptive schedule

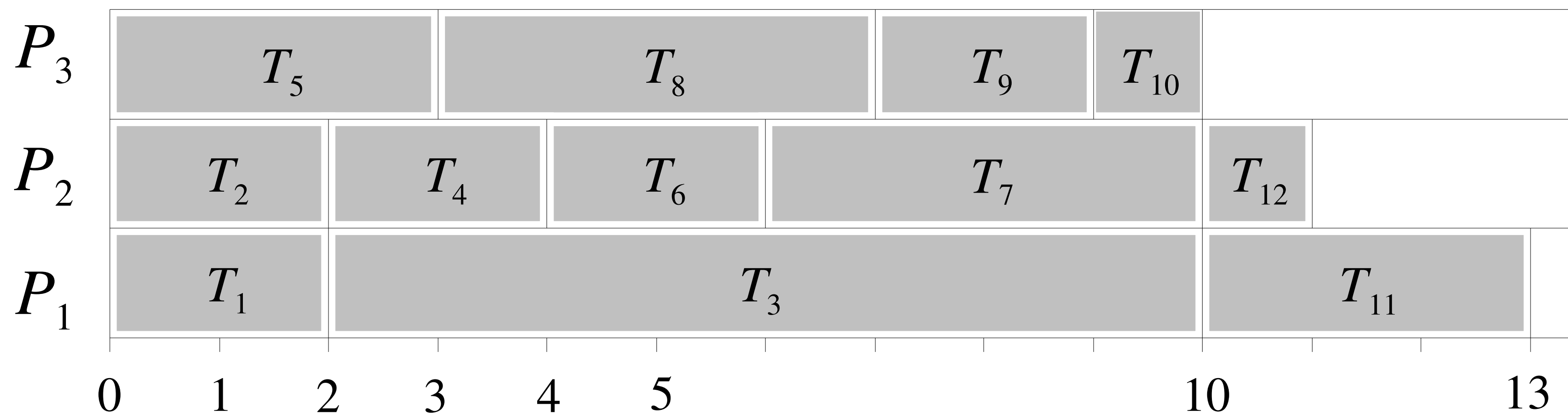
In the above example, let (2, 2, 8, 2, 3, 2, 4, 4, 2, 1, 3, 1) be the vector of processing times, and assume all release times = 0

Assume furthermore that there are m=3 identical processors ($\mathcal{P} = \{P_1, P_2, P_3\}$) available for processing the tasks

Gantt chart of a non-preemptive schedule:



The Scheduling Model. *Schedule representation*



The Scheduling Model. Deterministic Model

Given a schedule S, the following can be determined for each task T_j *:*

flow time $F_j := c_j - r_j$

lateness $L_j = c_j - d_j$

tardiness $D_j = \max\{c_j - d_j, 0\}$

tardy task $U_j = \begin{cases} 0 & \text{if } D_j = 0 \\ 1 & \text{else} \end{cases}$

Evaluation of schedules

Maximum makespan $C_{max} = max\{c_j \mid T_j \in \mathcal{T}\}$

Mean flow time $\bar{F} := (1/n) \sum F_j$

Maximum lateness $L_{max} = max\{L_j \mid T_j \in \mathcal{T}\}$

Mean tardiness $\bar{D} := (1/n) \sum D_j$

Given a set of tasks and a processor environment there are generally many possible schedules

Evaluating schedules: distinguish between *good* and *bad* schedules

This leads to different *optimization criteria*

Minimizing the maximum makespan C_{max}

C_{max} criterion: C_{max}-optimal schedules have minimum makespan

the total time to execute all tasks is minimal

Minimizing *schedule length* is important from the viewpoint of the owner of a set of processors (machines):
This leads to both, the maximization of the processor utilization factor (within schedule length C_{max}), and the minimization of the maximum in-process time of the scheduled set of tasks

Deadline related criteria

If deadlines are specified for (some of) the tasks we are interested in a schedule in which all tasks complete before their deadlines expire

Question: does there exist a schedule that fulfills all the given conditions?

Such a schedule is called *valid* (*feasible*)

Here we are faced in principle with a *decision problem*

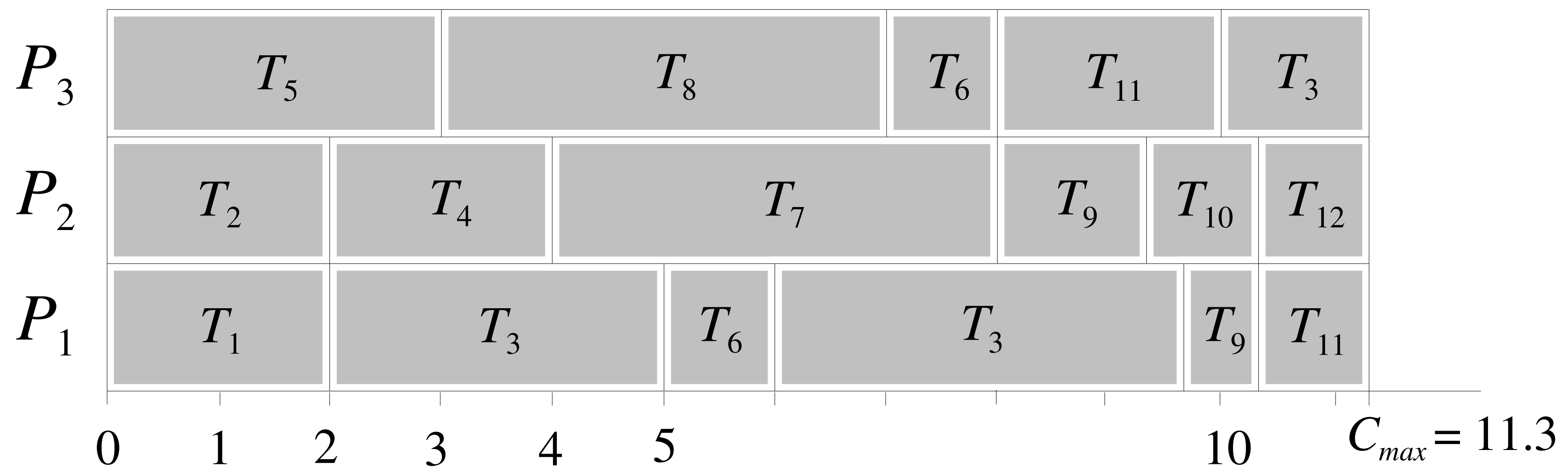
If, however, a valid schedule exits, we would of course like to get it explicitly

If a valid schedule exists we may wish to find a schedule that has certain additional properties, such as minimum makespan or minimum mean flow

Hence in deadline related problems we often additionally impose one of the other criteria

Example 3

Gantt chart of a preemptive schedule:



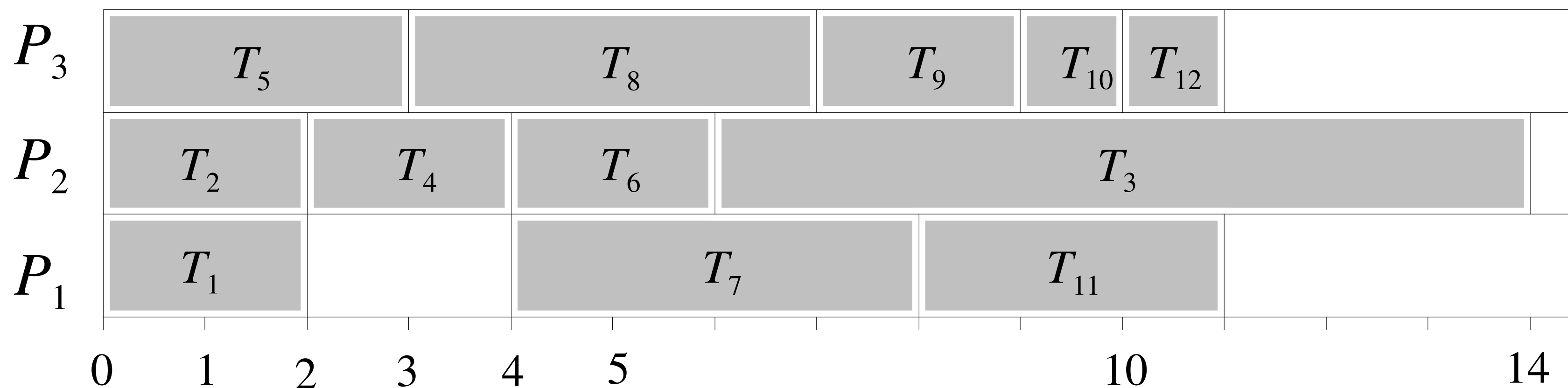
(1) In the schedule of example the flow time of tasks

$F(T_1)=2, F(T_2)$ = 2,F(T_3)=???, etc.

Example 4: non-preemptive schedule with due dates

For the task set as specified before, let in addition due dates be given by the vector (8, 2, 16, 4, 4, 8, 8, 8, 10, 8, 10, 11).

In the schedule below, task T_{10} with due date 8 violates its due date by two time units.



(2) In the schedule of example task T_{10} has lateness ???; for all other tasks L_j is less equal ???.

The tardiness of T_{10} = ????, and it is ??? for all other tasks;

hence $U_{10} = ???$, and $U_j = ???$ for all other tasks

(3) In the same schedule task T_1 has earliness ??, T_2 has earliness ??, etc.

Examples

(1) In the schedule of example 3 the flow time of tasks T_1 and T_2 is 2, that of T_3 is 11.3, etc.

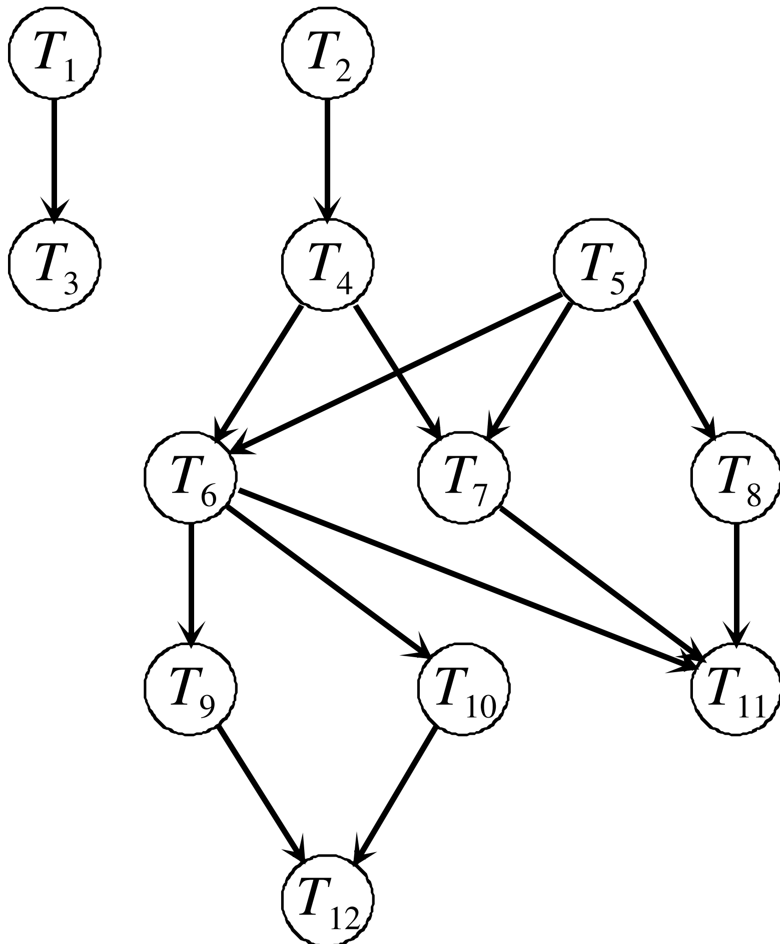
(2) In the schedule of example 4 task T_{10} has lateness 2; for all other tasks L_j is less than or equal 0. The tardiness of T_{10} , and it is 0 for all other tasks; hence $U_{10} = 1$, and $U_j = 0$ for all other tasks

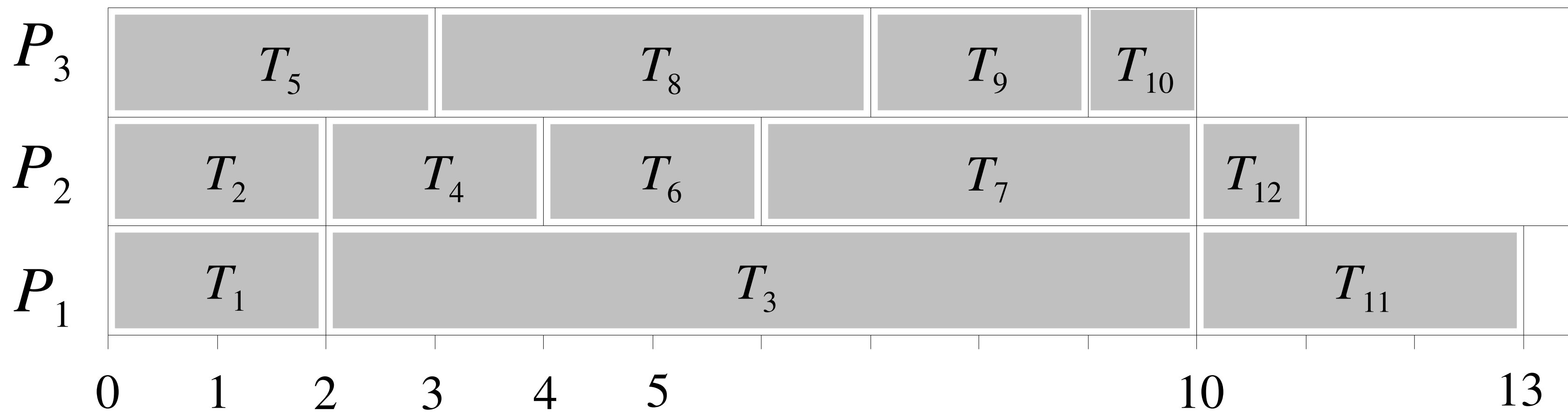
(3) In the same schedule task T_1 has earliness 6, T_2 has earliness 0, etc.

Example: Consider the task set as in Example 1, with processing times and due dates as specified in the respective Examples 2 and 3.

m=3, processing times (2, 2, 8, 2, 3, 2, 4, 4, 2, 1, 3, 1),

due dates (8, 2, 16, 4, 4, 8, 8, 8, 10, 8, 10, 11).




P_3
P_2
P_1
T_5
T_8
T_9
T_{10}
T_2
T_4
T_6
T_7
T_{12}
T_1
T_3
T_{11}
0
1
2
3
4
5
10
13

Topic 2
Scheduling on Parallel Processors

2.1 Minimizing Schedule Length
- Identical Processors
- Uniform Processors

2.2 Minimizing Mean Flow Time
- Identical Processors
- Uniform Processors

2.3 Minimizing Due Date Involving Criteria
- Identical Processors
- Uniform Processors

Independent tasks

Identical Processors $P \mid\mid C_{max}$

The first problem considered is $P \mid\mid C_{max}$ where

- set of independent tasks
- identical processors
- minimize schedule length.

Complexity analysis:

- the problem is not easy to solve since even simple cases such as scheduling on two processors can be proved to be NP-hard.

There is no hope of finding a polynomial time algorithm

Solution: find an approximation algorithm for the original problem and evaluate its worst case as well as its mean behavior.

One of the most often used general approximation strategies for solving scheduling problems is *list scheduling*

- priority list of the tasks is given
- at each step the first available processor is selected to process the first available task on the list

The accuracy of a given list scheduling algorithm depends on the order in which tasks appear on the list.

Identical Processors. List Scheduling

$W_{seq} = \sum_{i=1}^{n} p_i$ be the total work of all jobs

$p_{\max}$ is the maximum processing time of a job.

W_{idle} be the total idle intervals, $W_{idle} \leq p_{\max}(m-1)$

$C \leq \frac{W_{seq} + W_{idle}}{m}$ is the completion time of the set of tasks.

$$C \leq \frac{W_{seq} + p_{\max}(m-1)}{m} \leq \frac{W_{seq} - p_{\max}}{m} + p_{\max}, \quad C \leq \frac{W_{seq}}{m} + \frac{(m-1)}{m} p_{\max}$$

$\frac{W_{seq}}{m}$ and $p_{\max}$ are lower bounds of C_{opt}^{seq}, it follows that the worst-case performance bound is $\rho^{seq} \leq 2 - \frac{1}{m}$.

Identical Processors. *LPT Algorithm for $P \mid\mid C_{max}$*

Approximation algorithm for $P \mid\mid C_{max}$:

One of the simplest algorithms is the *LPT algorithm* in which the tasks are arranged in order of non-increasing p_j .

```
Algorithm  LPT for P || Cmax .
begin
Order tasks such that p1 ≥ ··· ≥ pn ;
for i = 1 to m do si := 0;
    -- processors Pi are assumed to be idle from time si = 0 on
j := 1;
repeat
    sk := min{ si };
    Assign task Tj to processor Pk at time sk;
      -- the first non-assigned task from the list is scheduled on the first processor that
becomes free
    sk := sk + pj;  j := j + 1;
until j = n;       -- all tasks have been scheduled
end;
```

Identical Processors. *LPT Algorithm for* $P \mid\mid C_{max}$

Theorem *If the LPT algorithm is used to solve problem* $P \mid\mid C_{max}$, *then* $R_{LPT} = \frac{4}{3} - \frac{1}{3m}$

.□

an example showing that this bound can be achieved.

Let $n = 2m + 1$, $p = [2m - 1, 2m - 1, 2m - 2, 2m - 2, \ldots, m + 1, m + 1, m, m, m]$.

For $m = 3$, Next figure shows two schedules, an optimal one and an *LPT* schedule.

Identical Processors. *LPT Algorithm for P | | C_{max}*

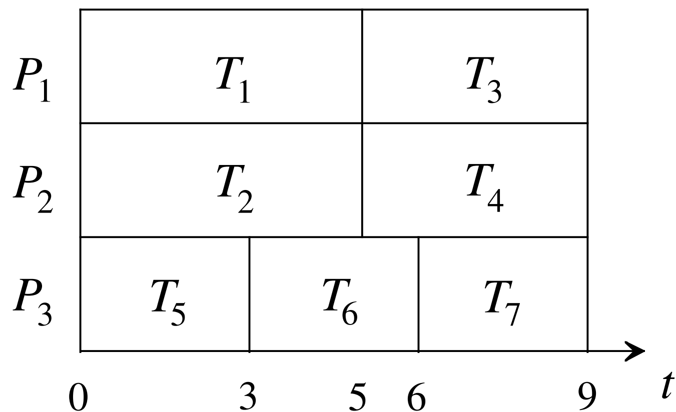
Example: $m = 3$ identical processors; $n = 2m + 1$,

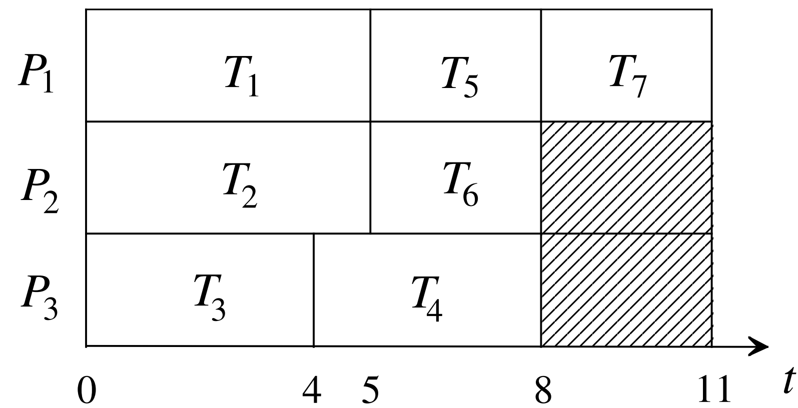
$p = [2m - 1, 2m - 1, 2m - 2, 2m - 2, \ldots, m + 1, m + 1, m, m, m]$.

Time complexity of this algorithm is $O(n\log n)$

- the most complex activity is to sort the set of tasks.

For $m = 3$, $p = [5, 5, 4, 4, 3, 3, 3]$.





(a) *an optimal schedule,* (b) *LPT schedule.*

Identical Processors. *LPT Algorithm for* $P\,|\,|\,C_{max}$

Example: $n = (m-1)m + 1$, $p = [1, 1, \ldots, 1, 1, m]$, $\prec$ is empty,

$L = (T_n, T_1, T_2, \ldots, T_{n-1})$, $L' = (T_1, T_1, \ldots, T_n)$.

The corresponding schedules for $m = 4$

	0–1	1–2	2–3	3–4
P_1	T_{13}			
P_2	T_1	T_4	T_7	T_{10}
P_3	T_2	T_5	T_8	T_{11}
P_4	T_3	T_6	T_9	T_{12}

t: 0 1 2 3 4

(a) an optimal schedule,

	0–1	1–2	2–3	3–7
P_1	T_1	T_5	T_9	T_{13}
P_2	T_2	T_6	T_{10}	
P_3	T_3	T_7	T_{11}	
P_4	T_4	T_8	T_{12}	

t: 0 1 2 3 7

(b) an approximate schedule

© 2014 A. Tchernykh. CIMAT 2014

Preemptions

Identical Processors, $P \mid pmtn \mid C_{max}$

Problem $P \mid pmtn \mid C_{max}$

- relax some constraints imposed on problem $P \mid \mid C_{max}$ and allow preemptions of tasks.
- It appears that problem $P \mid pmtn \mid C_{max}$ can be solved very efficiently.

It is easy to see that the length of a preemptive schedule cannot be smaller than the maximum of two values:

- the maximum processing time of a task and
- the mean processing requirement on a processor:

The following algorithm given by McNaughton (1959) constructs a schedule whose length is equal to C^*_{max}.

$$C^*_{max} = \max\{\max_j\{p_j\}, \frac{1}{m}\sum_{j=1}^{n} p_j\} .$$

© 2014 A. Tchernykh. CIMAT 2014

Identical Processors, $P \mid pmtn \mid C_{max}$ю *McNaughton's rule*

Algorithm *McNaughton's rule for* $P \mid pmtn \mid C_{max}$

begin

$C^*_{max} := max\{\Sigma_{j=1}^{n} p_j/m, max\{p_j \mid j = 1,...,n\}\}$; -- min schedule length

$t := 0$; $i := 1$; $j := 1$;

repeat

 if $t + p_j \leq C^*_{max}$

 then begin

  Assign task T_j to processor P_i , starting at time t;

  $t := t + p_j$; $j := j + 1$;

   -- assignment of the next task continues at time $t + p_j$

  end

 else begin

  Starting at time t, assign task T_j for $C^*_{max} - t$ units to P_i ;

   -- task T_j is preempted at time C^*_{max},

   -- assignment of T_j continues on the next processor at time 0

  $p_j := p_j - (C^*_{max} - t)$; $t := 0$; $i := i + 1$;

  end;

```
until j = n ;        -- all tasks have been scheduled
end;
```

Remarks: The algorithm is optimal. Its time complexity is $O(n)$

Question of practical applicability:

- preemptions are not free of cost (delays)
- two kinds of preemption costs have to be considered: time and finance.
  - Time delays are not crucial if the delay caused by a single preemption is small compared to the time the task continuously spends on the processor
  - Financial costs connected with preemptions, on the other hand, reduce the total benefit gained by preemptive task execution; but again, if the profit gained is large compared to the losses caused by the preemptions the schedule will be more useful and acceptable.

Precedence constraints

Identical Processors, $P \mid prec \mid C_{max}$

Given: task set $\mathcal{T}$ with

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

Let C_{max} be the length of the list schedule

Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

The above parameters can be changed:

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

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

List scheduling algorithms have unexpected behavior:

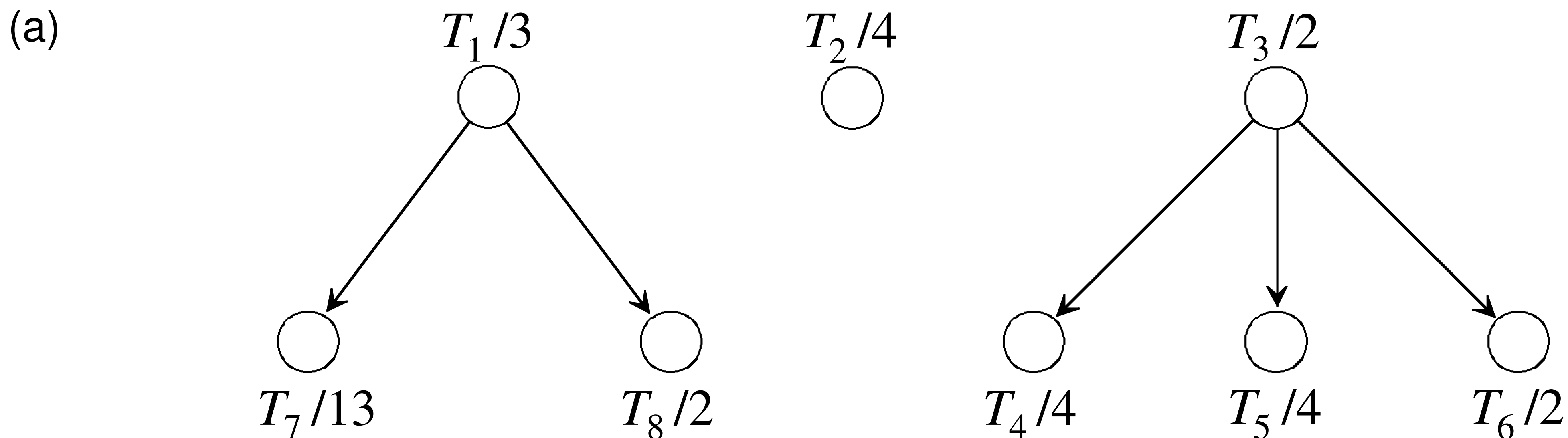
Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

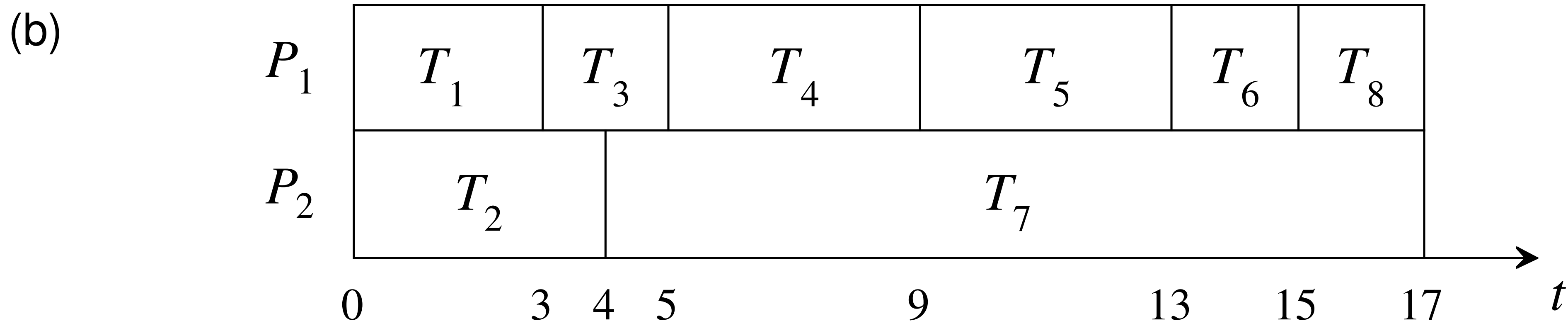
- the schedule length for problem $P \mid prec \mid C_{max}$
-

may increase

if:

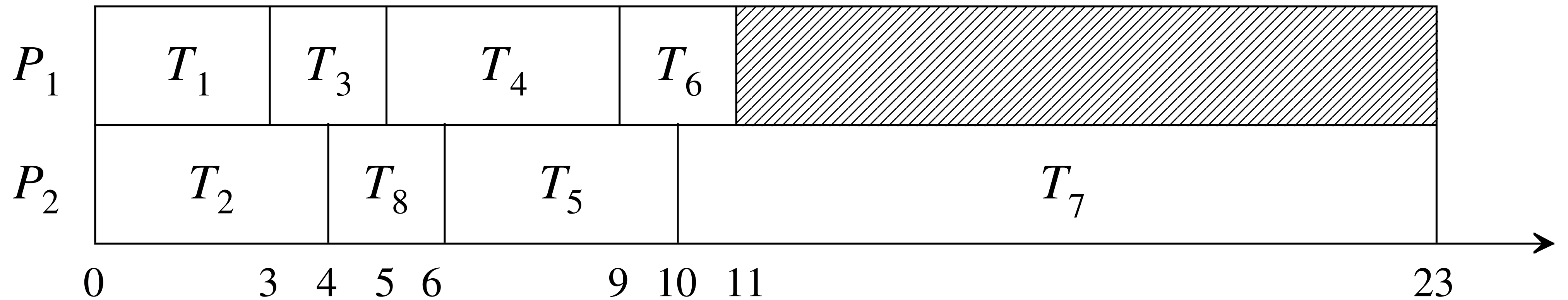
- the number of processors *increases*,
- task processing times *decrease*,
- precedence constraints are *weakened*, or
- the priority list changes



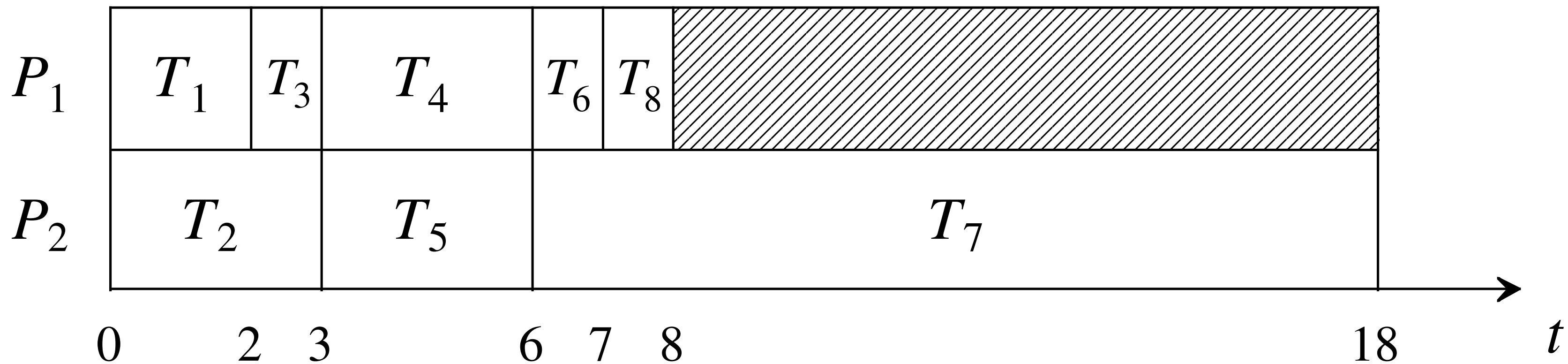


(a) *A task set,* $m = 2$, $L = (T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8)$,
(b) *an optimal schedule*

Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

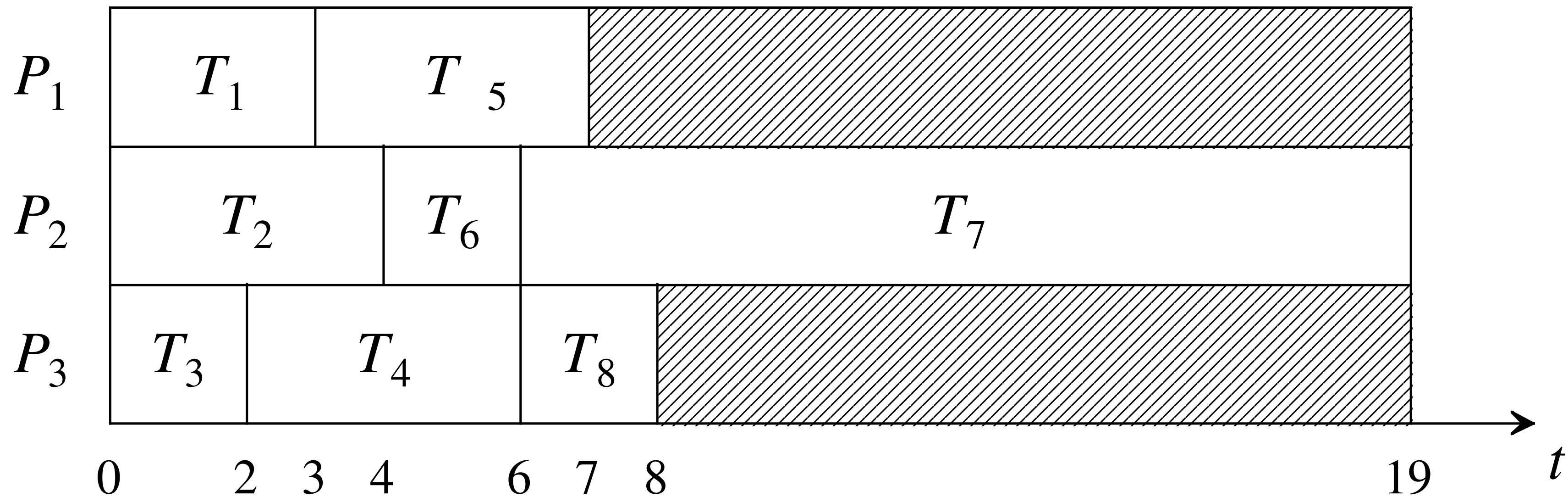


A new list $L' = (T_1, T_2, T_3, T_4, T_5, T_6, T_8, T_7)$.



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

Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies



Number of processors increased, $m = 3$

Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

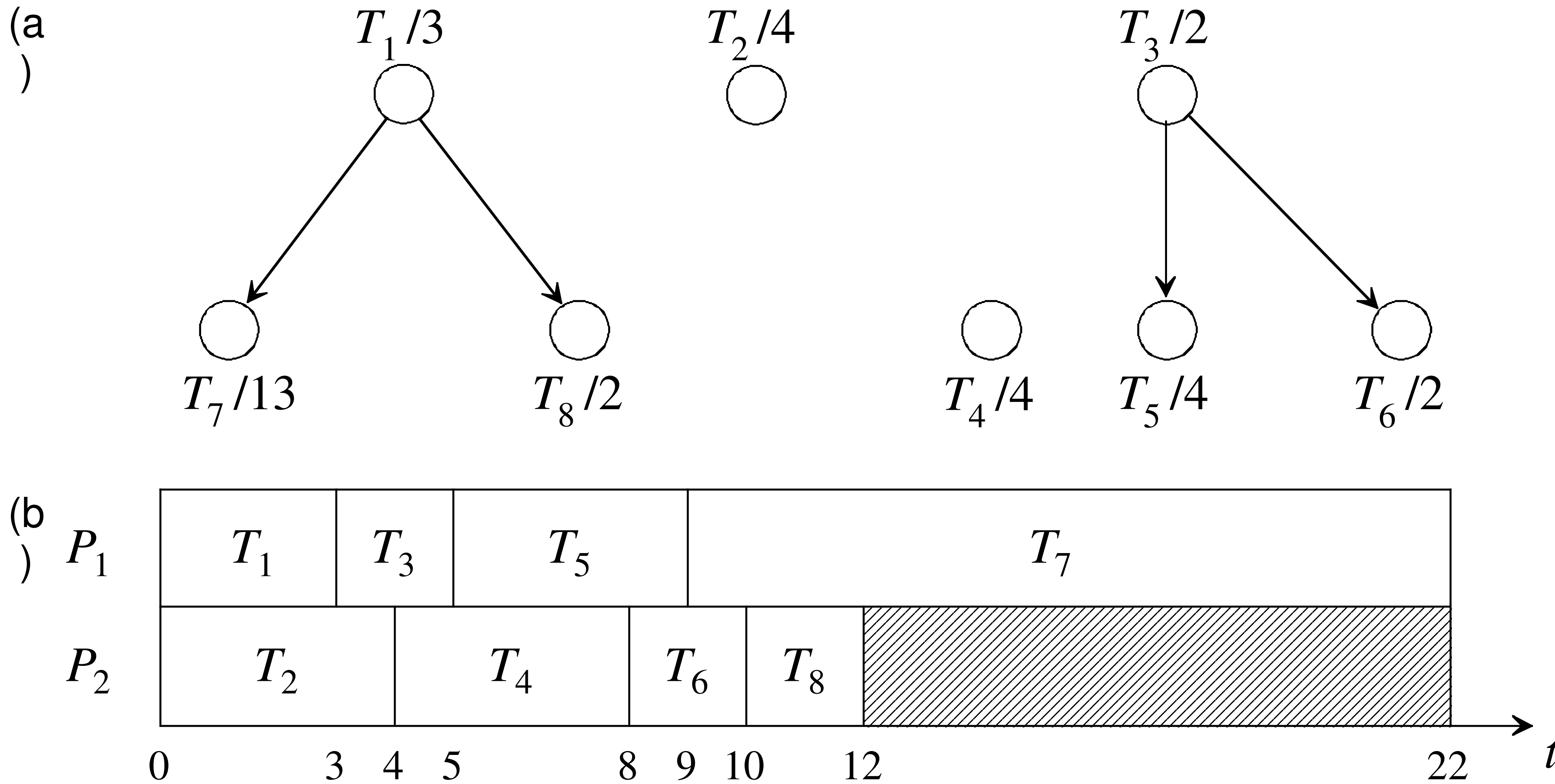


Figure 4-6 (a) *Precedence constraints weakened*, (b) *resulting list schedule.*

Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

These list scheduling anomalies have been discovered by Graham [Gra66], who has also evaluated the maximum change in schedule length that may be induced by varying one or more problem parameters.

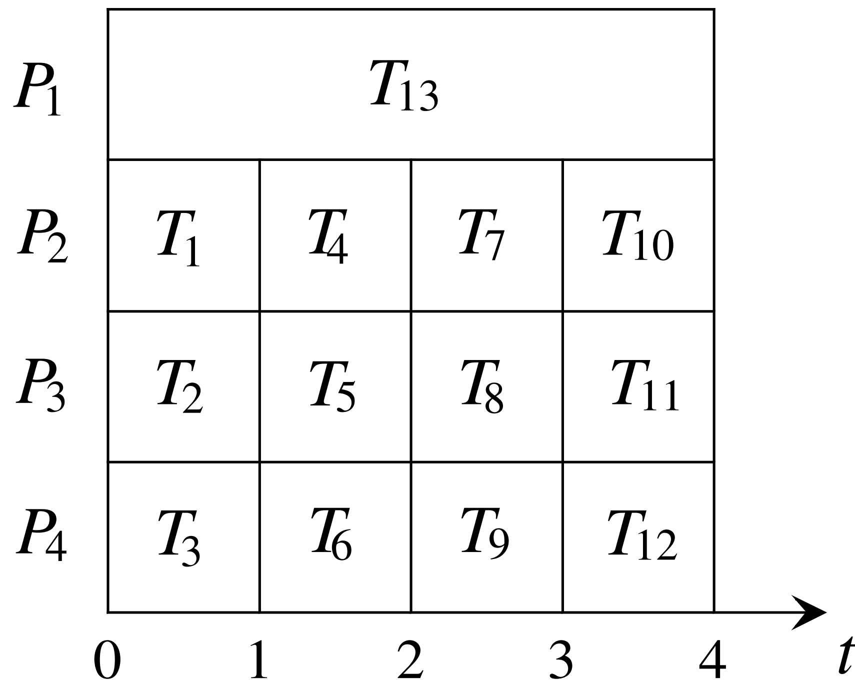
- Let the processing times of the tasks be given by vector p,
- let T be scheduled on m processors using list L, and
- let the obtained value of schedule length be equal to C_{max}.

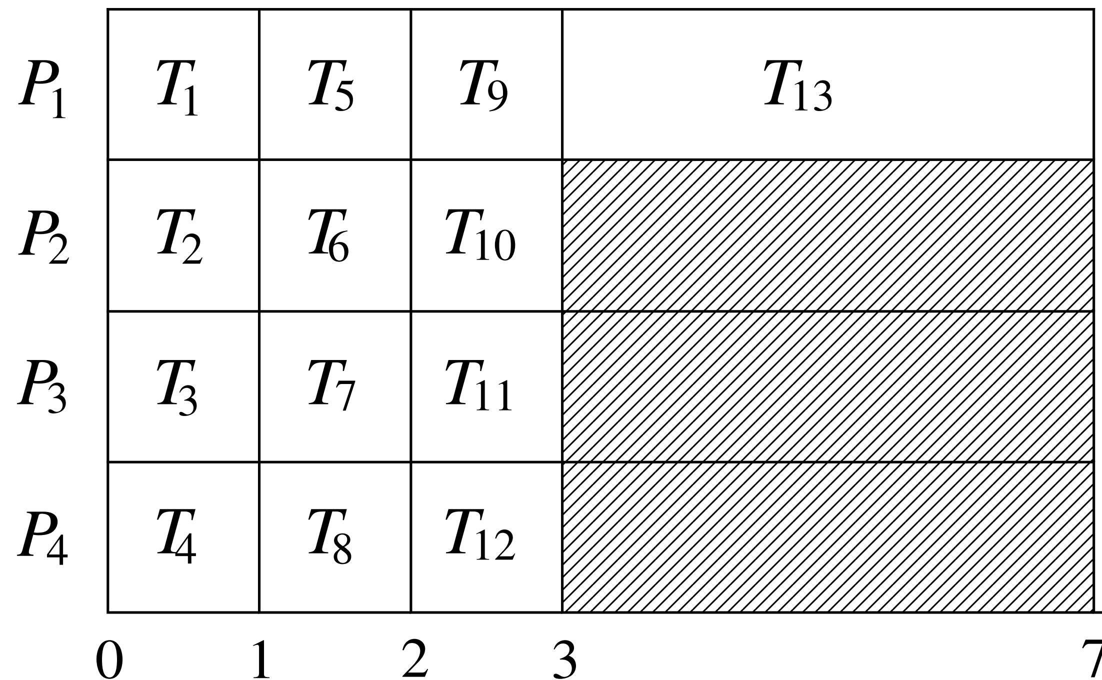
On the other hand, let the above parameters be changed:

- a vector of processing times $p' \leq p$ (for all the components),
- relaxed precedence constraints $\prec' \subseteq \prec$,
- priority list L' and the number of processors m'.
- Let the new value of schedule length be C'_{max} .

Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

Corollary (Graham 1966) *For an arbitrary list scheduling algorithm LS for* $P \mid\mid C_{max}$ *we have* $R_{LS} \le 2 - \frac{1}{m}$ if $m' = m$.





Schedules for Corollary
(a) *an optimal schedule,*
(b) *an approximate schedule.*

Topic 3
Scheduling on Parallel Processors

2.1 Minimizing Schedule Length
- Identical Processors
- Uniform and Unrelated Processors

2.2 Minimizing Mean Flow Time
- Identical Processors
- Uniform and Unrelated Processors

3 Minimizing Due Date Involving Criteria
- Identical Processors
- Uniform and Unrelated Processors

Model

Arrival time (or *release* or *ready time*) r_j … is the time at which task T_j is ready for processing

if the arrival times are the same for all tasks from $\mathcal{T}$, then $r_j = 0$ is assumed for all tasks

– *Due date* d_j … specifies a time limit by which T_j should be completed

problems where tasks have due dates are often called "soft" real-time problems. Usually, penalty functions are defined in accordance with due dates

– *Penalty functions* G_j define penalties in case of due date violations

– *Deadline* $\tilde{d}_j$ … "hard" real time limit, by which T_j must be completed

– *Weight* (*priority*) w_j ... expresses the relative urgency of T_j

Identical Processors. Deadline Criteria $P \mid r_j, \tilde{d}_j \mid -$

If deadlines are given:

- check if a feasible schedule exists (*decision problem*)

Single processor problem $P1 \mid p_j = 1, d_j \mid -$ can be solved in polynomial time

EDF algorithm is optimal

More than one processor: most problems are known to be NP-complete

The problems

$P \mid p_j = 1, d_j \mid -$ and $P \mid prec, p_j \in \{1, 2\}, d_j \mid -$

are NP-complete

Algorithmic approaches:

- exhaustive search
- heuristic algorithms
- approximation algorithms

© 2014 A. Tchernykh. CIMAT 2014

Identical Processors. Deadline Criteria $P \mid r_j, \tilde{d}_j \mid -$

Scheduling strategies:

A strategy is called "feasible", if the algorithm generates schedules where all tasks observe their deadlines (assuming this is actually possible)

three interesting deadline scheduling strategies:

EDF	Earliest Deadline First scheduling
LL	Least Laxity scheduling

Identical Processors. Deadline Criteria $P \mid r_j, \tilde{d}_j \mid -$

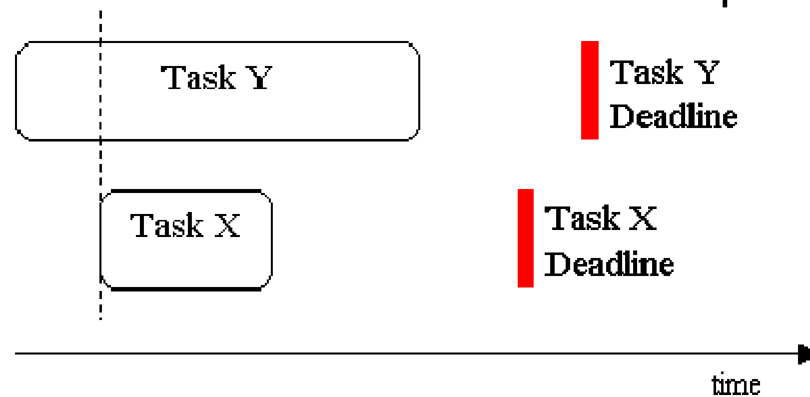
Earliest Deadline First Scheduling Policy

- means that the task that has the earliest deadline (task that has to be processed first) is to be scheduled next.
- EDF scheduler views task deadlines as more important than task priorities.
- Experiments have shown that the earliest deadline first policy is the most fair scheduling algorithm.

Identical Processors. Deadline Criteria $P \mid r_j, \tilde{d}_j \mid -$

More complex deadline scheduler is the “Least Laxity” (or “LL”) scheduler.

- takes into account both a task’s deadline and its processing load,



EDF deadline scheduler would allow Task X to run before Task Y, even if Task Y normally has higher priority.

- However, it could cause Task Y to miss its deadline.
- So perhaps an “LL” scheduler would be better

Identical Processors. Deadline Criteria $P \mid r_j, \tilde{d}_j \mid -$

Laxity is the value that describes how much computation there is still left before the deadline of the task if it ran to completion immediately. Laxity of a task is a measure for it's urgency.

Laxity = (Task Deadline – (Current schedule time + Rest of Task Exec. Time).

LL=D-t-Prest

It is the amount of time that the scheduler can “play with” before causing the task to fail to meet its deadline.

Least Laxity Scheduling Policy: the task that has the smallest laxity (meaning the least computation left before it's deadline) is scheduled next.

Thus, a Least Laxity deadline scheduler takes into account both deadline and processing load.

Identical Processors. Deadline Criteria $P \mid r_j, \tilde{d}_j \mid -$

Example: Comparison of strategies

Set of independent tasks: T = $\{T_1, T_2, ..., T_6\}$

Tasks: (*deadline*, *total execution time*, *arrival time*):

$T_1 = (5, 4, 0)$, $T_2 = (6, 3, 0)$, $T_3 = (7, 4, 0)$,

$T_4 = (12, 9, 2)$, $T_5 = (13, 8, 4)$, $T_6 = (15, 12, 2)$

Execution on *three identical processors:*

EDF-schedule (no preemptions): total execution time is 16

least laxity schedule (with preemptions): $\leq$ 8 preemptions, total execution time is 15

optimal schedule with 3 preemptions, total execution time = 15

Execution on a *single*, *three times faster processor*:

possible with no preemptions; total execution time is 40/3

Hence: a larger number of processors is not necessarily advantageous

Identical Processors. Deadline Criteria $P \mid pmtn, r_j, \tilde{d}_j \mid -$

Feasibility testing of problem $P \mid pmtn, r_j, \tilde{d}_j \mid -$ is done by applying a network flow approach (Horn 1974)

Given an instance of $P \mid pmtn, r_j, \tilde{d}_j \mid -$,

let $e_0 < e_1 < \ldots < e_k$, $k \le 2n-1$ be the ordered sequence of release times and deadlines together (e_i stands for r_j or $\tilde{d}_j$) (time intervals)

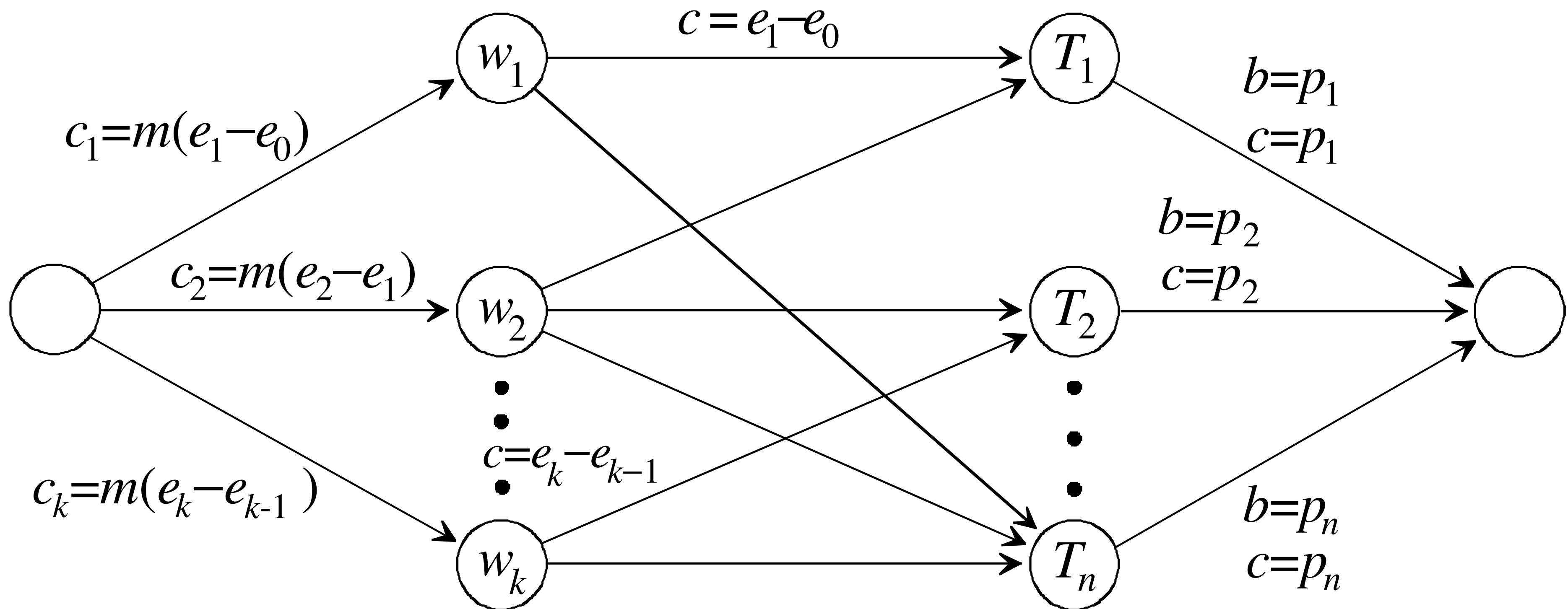
Construct a network with source, sink and two sets of nodes (Figure):

the first set (nodes w_i) corresponds to time intervals in a schedule;

node w_i corresponds to interval $[e_{i-1}, e_i]$, $i = 1, 2, \ldots, k$

the second set corresponds to the tasks

Identical Processors. Deadline Criteria $P \mid pmtn, \; r_j, \tilde{d}_j \mid -$



Identical Processors. Deadline Criteria $P \mid pmtn, r_j, \tilde{d}_j \mid -$

Flow conditions:

– The capacity of an arc joining the source to node w_i is $m(e_i - e_{i-1})$

- this corresponds to the total processing capacity of m processors in this interval

– If task T_j is allowed to be processed in interval $[e_{i-1}, e_j]$ then w_i is joined to T_j by an arc of capacity $e_i - e_{i-1}$

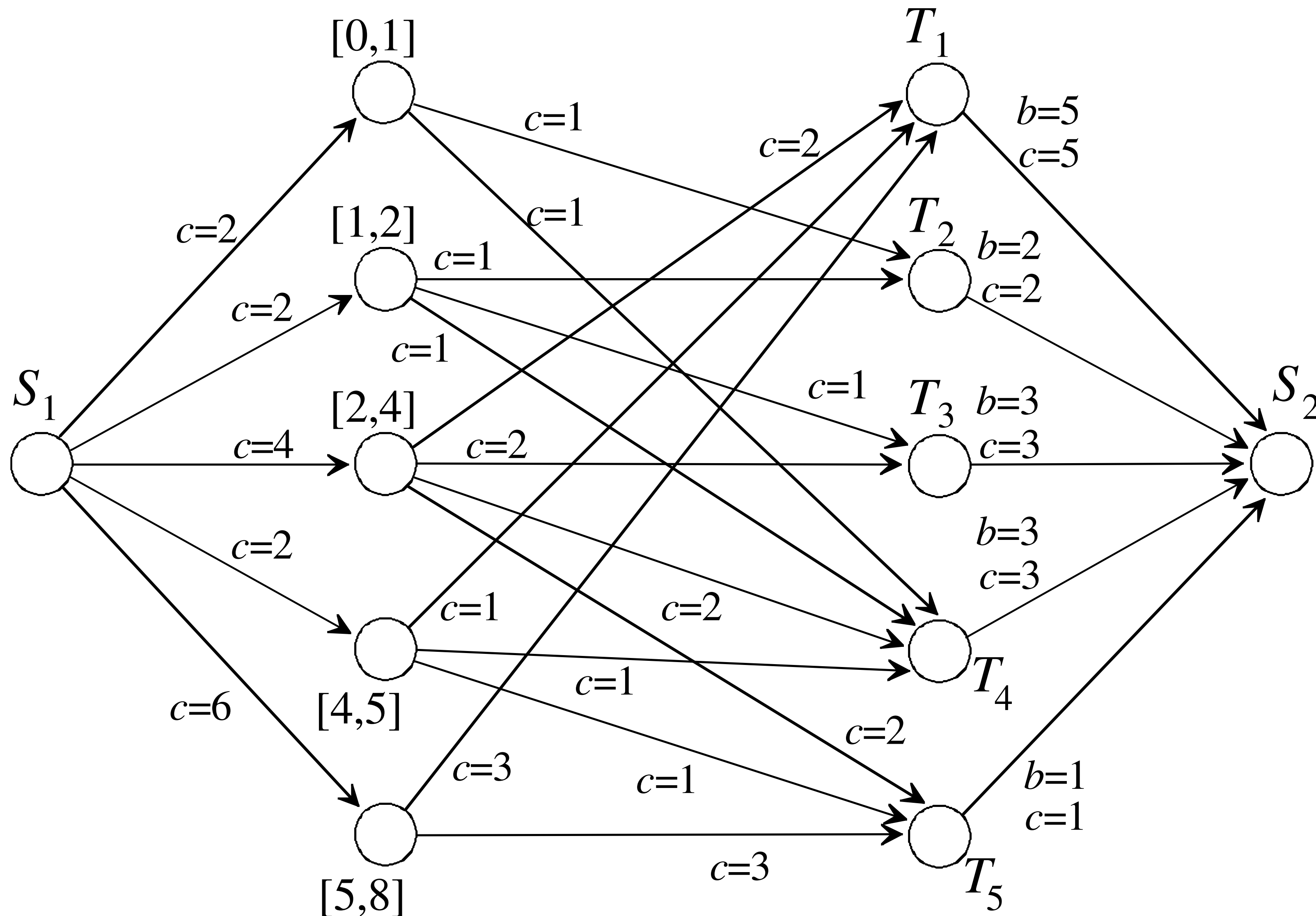
– Node T_j is joined to the sink of the network by an arc with lower and upper capacity equal to p_j

Finding a feasible flow pattern corresponds to constructing a feasible schedule; this test can be made in $O(n^3)$ time

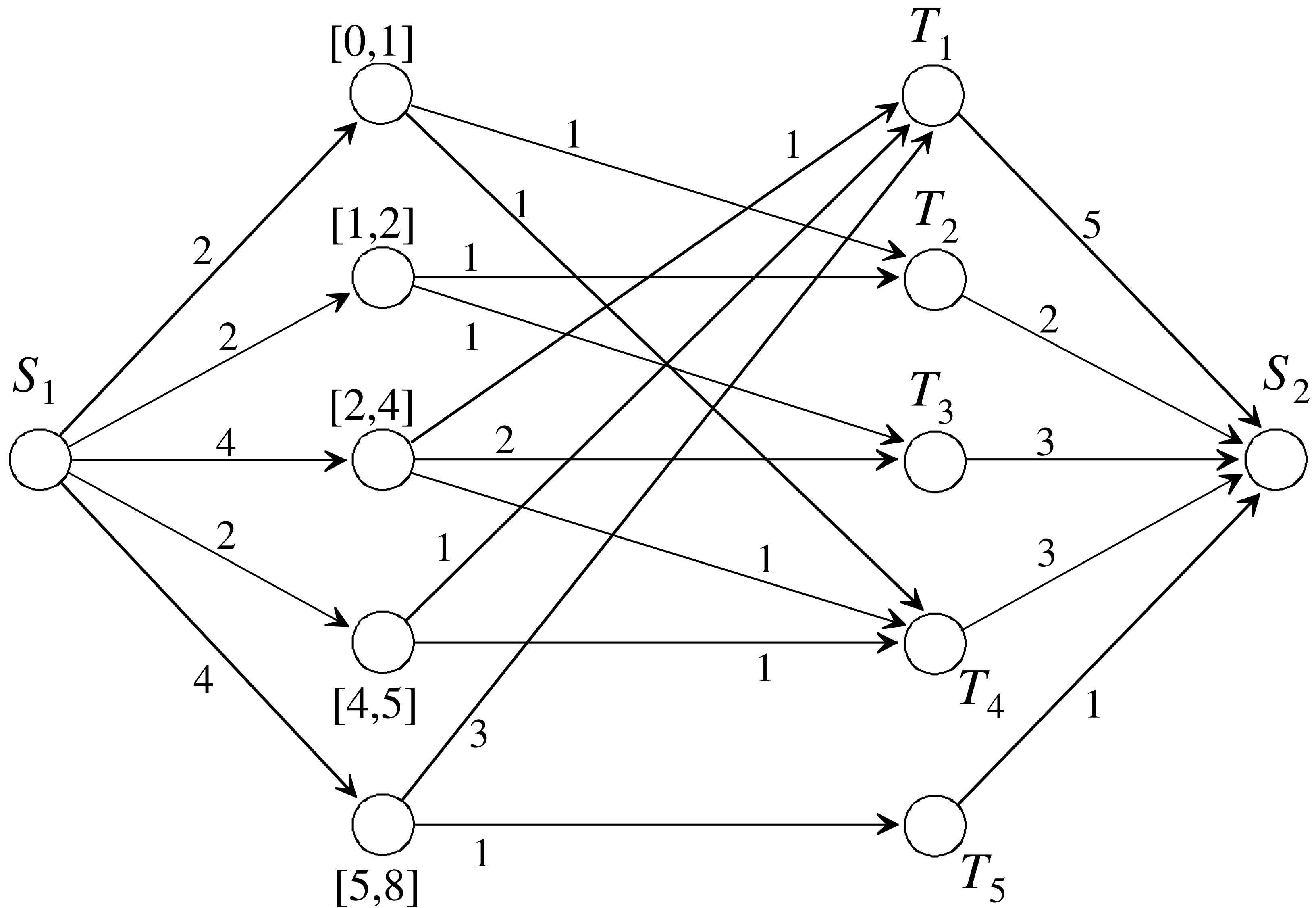
the schedule is constructed on the basis of the flow values on arcs between interval and task nodes.

© 2014 A. Tchernykh. CIMAT 2014

Example. $n = 5$, $m = 2$, $p = [5, 2, 3, 3, 1]$, $r = [2, 0, 1, 0, 2]$, and $d = [8, 2, 4, 5, 8]$.

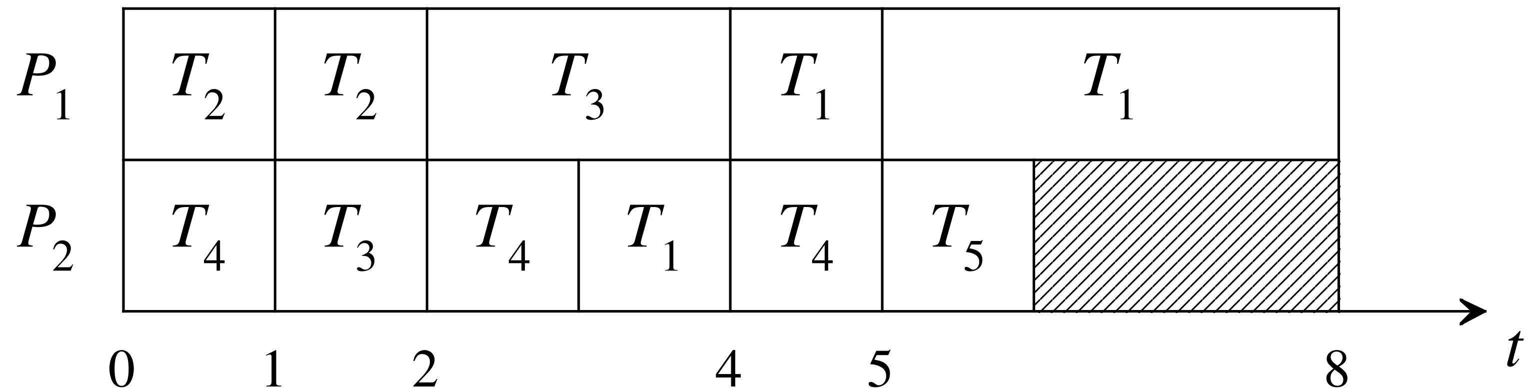


(a) corresponding network



(b) feasible flow pattern

(c) optimal schedule



Scheduling on Parallel Processors

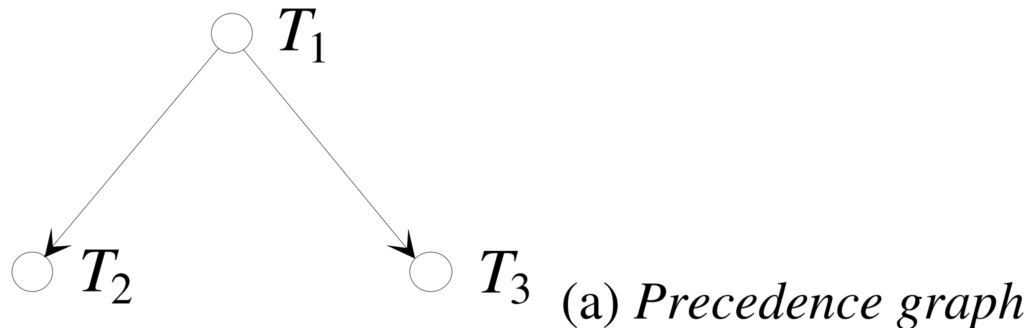
Communication Delays and Multiprocessor Tasks

- Introductory Remarks
- Scheduling Multiprocessor Tasks
  - Parallel Processors
  - Refinement Scheduling
- Scheduling Uniprocessor Tasks with Communication Delays
  - Scheduling without Task Duplication
  - Scheduling with Task Duplication
  - Considering Processor Network Structure
- Scheduling Divisible Tasks

Scheduling Uniprocessor Tasks with Communication Delays

The following simple example serves as an introduction to the problems.

Let there be given three tasks with precedences as shown in Figure (a).



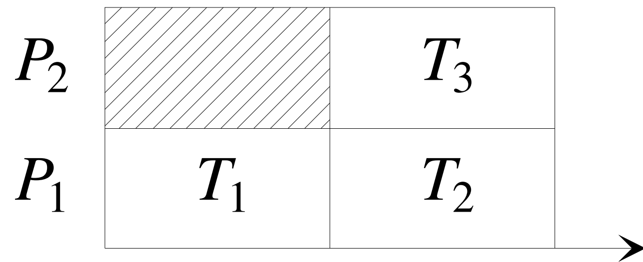
(a) *Precedence graph*

The computational results of task T_1 are needed by both successor tasks, T_2 and T_3 .

We assume unit processing times.

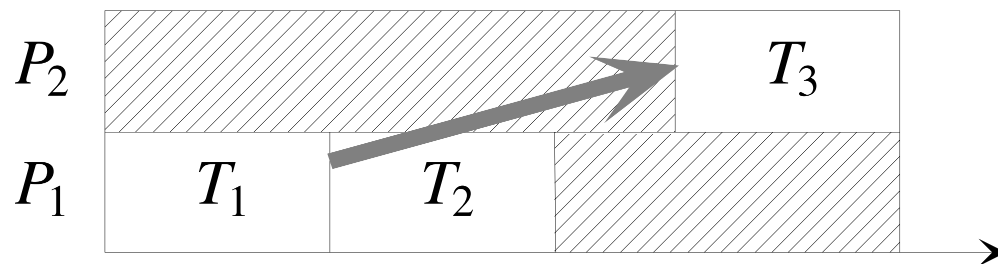
For task execution there are two identical processors, connected by a communication link.
To transmit the results of computation T_1 along the link takes 1.5 units of time.

Scheduling Uniprocessor Tasks with Communication Delays



(b) *Schedule without consideration of communication delays*

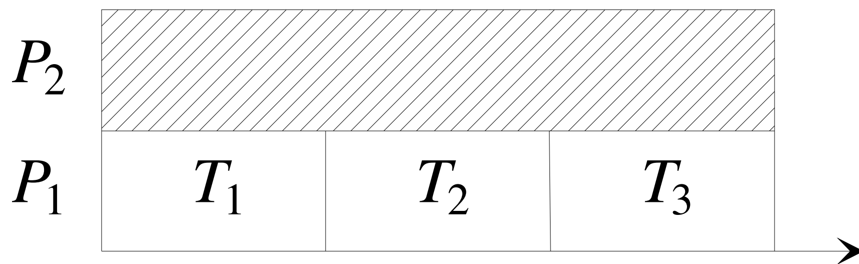
The schedule in Figure (b) shows a schedule where communication delays are not considered.



(c) *Schedule considering communication from* T_1 *to* T_3

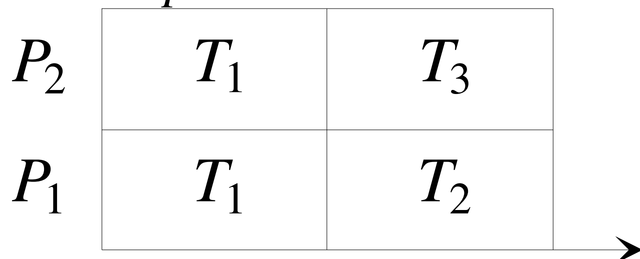
The schedule (c) is obtained from (b) by introducing a communication delay between T_1 and T_3 .

Schedule (d) demonstrates that there are situations where a second processor does not help to gain a shorter schedule.



(d) *Optimal schedule without task duplication*

The fourth schedule, (e), demonstrates another possibility: if task T_1 is processed on both processors, an even shorter schedule is obtained. The latter case is usually referred to as *task duplication*.



(e) *Optimal schedule with task duplication*

Scheduling Uniprocessor Tasks with Communication Delays

Communication delays are the same for all tasks

- so-called *uniform delay scheduling*.

Other approaches distinguish between *coarse grain* and *fine grain* parallelism:

- high computation-communication ratio can be expected in coarse grain parallelism.

As pointed out before, *task duplication* often leads to shorter schedules; this is in particular the case if the communication times are large compared to the processing times.

Bin Packing Problem

1. Introduction

Metaphorically, there never seem to be enough bins for all one needs to store.

Mathematics comes to the rescue with the *bin packing problem* and its relatives. The bin packing problem raises the following question:

- given a finite collection of n weights w_1, w_2, w_3, ... , w_n, and
- a collection of identical bins with capacity C (which exceeds the largest of the weights),
- what is the minimum number k of bins into which the weights can be placed without exceeding the bin capacity C?

© 2014 A. Tchernykh. CIMAT 2014

Outline

How few bins are needed to store a collection of items?

This problem, known as the 1-dimensional bin packing problem, is one of many mathematical packing problems which are of both theoretical and applied interest.

It is important to keep in mind that "weights" are to be thought of as indivisible objects rather than something like oil or water.

For oil one can imagine part of a weight being put into one container and any left over being put into another container.

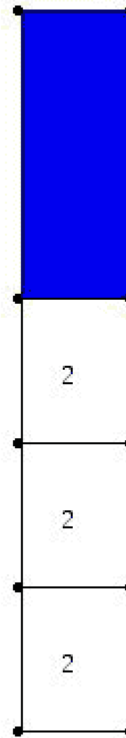
However, in the problem being considered here we are not allowed to have part of a weight in one container and part in another.

One way to visualize the situation is as a collection of rectangles which have height equal to the capacity C and a fixed width, whose exact size does not matter.

When an item is put into the bin it either falls to the bottom or is stopped at a height determined by the weights that are already in the bins.

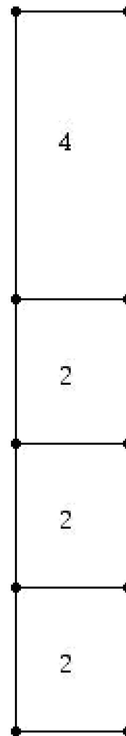
© 2014 A. Tchernykh. CIMAT 2014

The diagram below shows a bin of capacity 10 where three identical weights of size 2 have been placed in the bin, leaving 4 units of empty space, which are shown in blue.



Outline

By contrast with the situation above, the bin below has been packed with weights of size 2, 2, 2 and 4 in a way that no room is left over.



Basic ideas

The bin packing problem asks for the minimum number k of identical bins of capacity C needed to store a finite collection of weights w_1, w_2, w_3, ... , w_n so that no bin has weights stored in it whose sum exceeds the bin's capacity.

Traditionally

- capacity C is chosen to be 1 and
- weights are real numbers which lie between 0 and 1,
- for convenience of exposition, C is a positive integer and the weights are positive integers which are less than the capacity.

Example 1:

- Suppose we have bins of size 10. How few of them are required to store weights of size 3, 6, 2, 1, 5, 7, 2, 4, 1, 9?

Basic ideas

The weights to be packed above have been presented in the form of a *list* L ordered from left to right.

For the moment we will seek procedures (algorithms) for packing the bins that are "driven" by a given *list* L and a capacity size C for the bins.

The goal of the procedures is to minimize the number of bins needed to store the weights.

A variety of simple ideas as to how to pack the bins suggest themselves.

One of the simplest approaches is called *Next Fit* (NF).

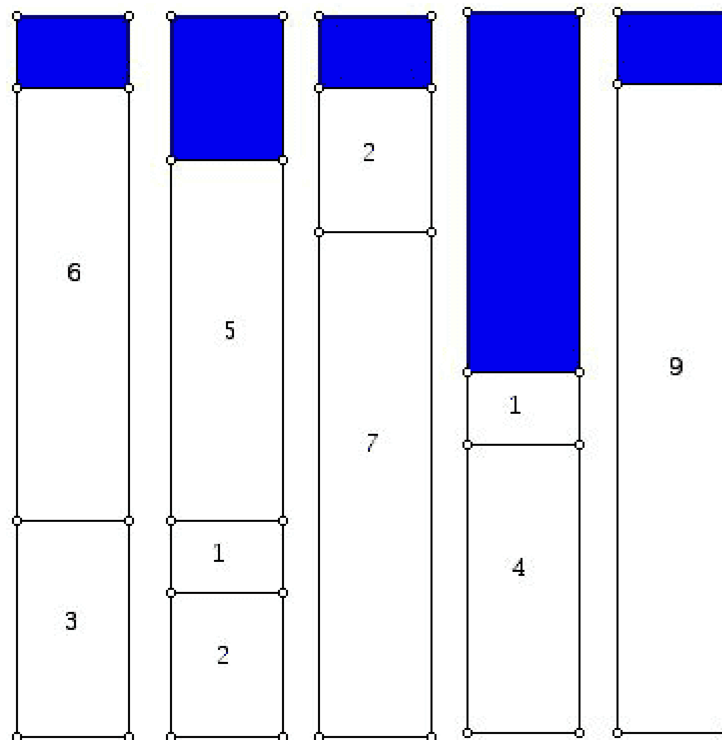
The idea behind this procedure is to open a bin and place the items into it in the order they appear in the list.

If an item on the list will not fit into the open bin, we close this bin permanently and open a new one and continue packing the remaining items in the list.

Basic ideas *Next Fit* (NF)

If some of the consecutive weights on the list exactly fill a bin, the bin is then closed and a new bin opened.

When this procedure is applied to the list above we get the packing shown below.



Basic ideas *Next Fit* (NF)

Next Fit is

- very simple,
- allows for bins to be shipped off quickly, because even if there is some extra room in a bin, we do not wait around in the hope that an item will come along later in the list which will fill this empty space.

One can imagine having a fleet of trucks with a weight restriction (the capacity C) and one packs weights into the trucks.

If the next weight can not be packed into the truck at the loading dock, this truck leaves and a new truck pulls into the dock.

We keep track of how much room remains in the bin open at that moment.

In terms of how much time is required to find the number of bins for *n* weights, one can answer the question using a procedure that takes a linear amount of time in the number of weights (*n*).

Clearly, NF does not always produce an optimal packing for a given set of weights. You can verify this by finding a way to pack the weights in Example 1 into 4 bins.

Basic ideas *Next Fit* (NF)

Procedures such as NF are sometimes referred to as *heuristics* or *heuristic algorithms* because although they were conceived as ways to solve a problem optimally, they do not always deliver an optimal solution.

Can we find a way to improve on NF so as to design an algorithm which will always produce an optimal packing?
A natural thought would be that if we are willing to keep bins open in the hope that we will be able to fill empty space with items later in list L, we will typically use fewer bins.

Basic ideas *First Fit* (FF)

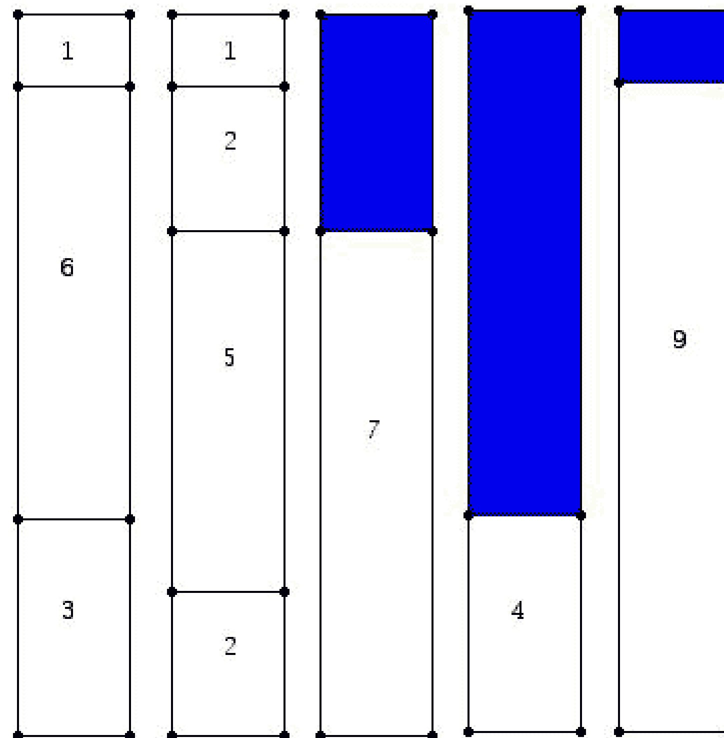
The simplest way to carry out this idea is known as *First Fit*.

We place the next item in the list into the first bin which has not been completely filled (thought of as numbered from left to right) into which it will fit.

- When bins are filled completely they are closed
- If an item will not fit into any currently open bin, a new bin is opened.

Basic ideas *First Fit* (FF)

- The result of carrying out First Fit for the list in Example 1 and with bins of capacity 10 is shown below:



Basic ideas *First Fit* (FF)

Both methods we have tried have yielded 5 bins.

We know that this is not the best we can hope for.

One simple insight is obtained by computing the total sum of the weights and dividing this number by the capacity of the bins.

Since we are dealing with integers, the number of bins we need must be at least

$\lceil \Omega / C \rceil$ where $\Omega = \sum_{i=1}^{n} w_i$.

(Note that $\lceil x \rceil$ denotes the smallest integer that is greater than or equal to *x*).

Clearly, the number of bins must always be an integer. In Example 1, since Ω is 40 and C is 10, we can conclude that there is hope of using only 4 bins.

However, neither Next Fit nor First Fit achieves this value with the list given in Example 1. Perhaps we need a better procedure.

Basic ideas *Best Fit* (BF) and *Worst Fit* (WF)

Two other simple methods in the spirit of Next Fit and First Fit have also been looked at.

These are known as *Best Fit* (BF) and *Worst Fit* (WF).

For Best Fit, one again keeps bins open even when the next item in the list will not fit in previously opened bins, in the hope that a later smaller item will fit.

The criterion for placement is that we put the next item into the currently open bin (e.g. not yet full) which leaves the least room left over. (In the case of a tie we put the item in the lowest numbered bin as labeled from left to right.)

For Worst Fit, one places the item into that currently open bin into which it will fit with the most room left over.

Basic ideas *Best Fit* (BF) and *Worst Fit* (WF)

The amount of time necessary to find the minimum number of bins using either FF, WF or BF is higher than for NF. What is involved here is $n \log n$ implementation time in terms of the number n of weights.
The distinction between First Fit, Best Fit and Worst Fit:

- suppose that we currently have only 3 bins open with capacity 10
- *remaining space* as follows:
  - Bin 4, 4 units,
  - Bin 6, 7 units, and
  - Bin 9 with 3 units.

Suppose the next item in the list has size 2.

First Fit puts this item in Bin 4, Best Fit puts it in Bin 9, and Worst Fit puts it in Bin 6!
One difficulty is that we are applying "good procedures" but on a "lousy" list. If we know all the weights to be packed in advance, is there a way of constructing a good list?

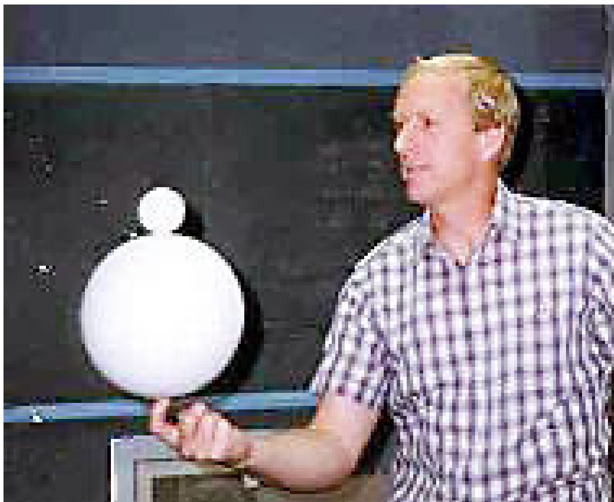
Basic ideas

Bin packing is

- a very appealing mathematical model, and yet work on this problem is surprisingly recent.
- about 35 years old.
- Major pioneers and contributors in working on this problem are

Edward Coffman

Michael Garey


Ronald Graham

David Johnson

More approaches to bin packing

All of the algorithms we have discussed so far have the property that the fact that there may be more items in the list to pack (farther to the right than one being currently worked on) does not affect what is done to pack the current item.

Such algorithms are known as *on-line*.

The idea for on-line algorithms, whether they are being used to solve bin packing problems or other combinatorial optimization problems, is that

- not all the components of the problem are known in advance

In the bin packing case one can imagine an industrial situation where items with different weights are being produced and then are being placed in bins which are at some stage to be shipped to customers.

More approaches to bin packing

In contrast to an on-line point of view is the possibility of using an *off-line* approach.

- one thinks of having all of the items to be packed in advance.
- one can ask for the given weights to be packed if there is some rearrangement of the weights into a list different from the original which might be used to give a better result for the number of bins required.

- If there are n items to pack, the number of potential lists for these n items is $n!$, since the first item can be chosen in n ways, the second in n-1, etc., giving $n!$ as the number of different possible lists.

- choosing a list for an optimal packing has the flavor of looking for a needle in a haystack. (Even for 20 items to pack, say, 20! is a very large number.)

More approaches to bin packing

In the on-line environment using FF, for example, at a given stage one may have so many open bins that it becomes economically unrealistic to have so many empty bins being monitored in the hope that later items will fit efficiently into them.

This suggests a version of bin packing where one seeks a packing heuristic but is limited to having at most ***K bins*** open at a given time.

These heuristics are known as ***bounded-space on-line heuristics***.

For the heuristics discussed above one can try to develop a *K open bin* bounded space version of the heuristic.

However, the situation for K > 1 open bins raises some tantalizing issues. Not only do we have the option of specifying the way the bins are packed, we have an option of specifying when a bin will be shut down (closed permanently) other than when it is completely full!

More approaches to bin packing

Suppose that we have K open bins and we now must pack a weight which does not fit into any of the open bins.

Since we must open a new bin, we have to choose an old bin to shut down.

This can be done either by

- picking the lowest numbered bin to shut down (this is in the spirit of FF) or by
- shutting down the bin which is closest to being completely full (this is in the spirit of BF).

Using either the FF or BF packing rule with the FF or BF closing rule, we get four(!) new heuristics in the K open bin environment.

More approaches to bin packing

In Example 1 we noticed that the very large item to pack at the end of the list made it necessary to have an extra bin opened at the end.

Intuitively, it seems like a good idea to try to pack large items first.

This is the same strategy you would use to pack a suitcase for a vacation; you would not leave a large-volumed item to the end!

This suggests the following approach to off-line bin packing.

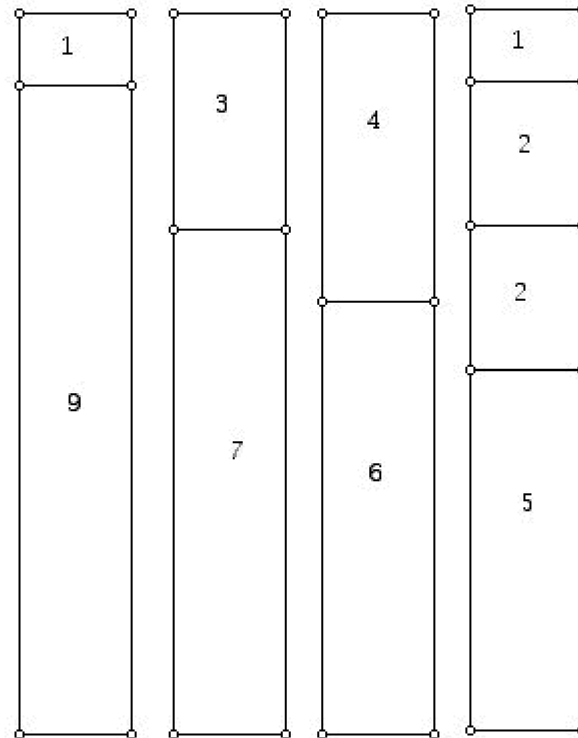
Choose the list where the weights appeared in sorted order, from largest to smallest. Now use any of the algorithms that we have discussed to carry out the packing.

We have already discussed four packing algorithms: NF, FF, BF, and WF and

We now have four new approaches which we will call NFD, FFD, BFD, and WFD where in each case the "D" refers to decreasing.

For example, FFD means First Fit Decreasing, and would give rise to the packing for the weights in Example 1 shown below. Note that this packing is optimal.

More approaches to bin packing



Not only can one not pack these weights into fewer bins but also there is no wasted space.

Of course, there are many situations where optimal packings still have unused space.

More approaches to bin packing

Among all packings of this kind one might look for that packing with various characteristics, say, that the fewest bins have extra space.

Alternatively, one might want a packing where the number of bins used is minimal but as many bins as possible have some extra room. (A little extra space might allow putting packing material into the bin to prevent breakage during shipment.)

Once one has an optimal packing, one can often either rearrange the items in one of the bins or between bins in a way that achieves some secondary goal beyond minimizing the number of bins.

The fact that FFD yielded an optimal solution in Example 1 does not mean that this will be the case for other problems the method is applied to.

Perhaps you are not surprised to learn that First Fit Decreasing does not always yield optimal solutions. Here is an example:

More approaches to bin packing

Example 2:

Suppose that one has bins of capacity 20.

What would be the fewest bins needed to pack weights of size 4, 8, 7, 10, 3, 8?

More approaches to bin packing

Since the sum of the weights is 40, with bins of capacity 20 a packing with only 2 bins might be achieved.

In fact there is indeed a packing needing only two bins:

- Bin 1: 8, 8, 4 and
- Bin 2: 10, 7, 3.

In this notation the item of weight 8 is at the bottom of Bin 1, next comes the item of weight 8, and the item of size 4 occupies the space at the top of the bin. Note that this solution is not unique because the items within the bin can be permuted.

However, with this particular list none of the procedures, NF, FF, BF, WF, NFD, FFD, BFD, or WFD yields an optimal number of bins.

In each case 3 bins are required, as you can check for yourself.

If one had been given, say, the list 8, 4, 8, 10, 7, 3 then NF, FF, BF, and WF would all give rise to an optimal packing (with the weights packed slightly differently from the solution given above).

Strip Packing Problem

The problem

- packing a set of *n* rectangles into an open-ended bin of
  - fixed width *C* and
  - infinite height.
  - the rectangles must not overlap each other
  - The idea is to pack the rectangles in a way that minimises the overall height of the bin.

The clear difference between this problem and the two-dimensional bin packing problem, is that there is only one bin, so instead of trying to minimise the number of bins used, the aim is to minimise the height of the single bin.

All the rectangles must be packed orthogonally. They cannot be rotated and they must be packed with their width parallel to the bottom of the bin.

The rectangles correspond to a set of tasks,

- heights being the amount of processing time they require
- widths the amount of contiguous memory (processors) they need.
- The width of the bin corresponds to the amount of memory (procs) available
- The aim is to schedule all of the tasks so that they are completed in the least possible time

Strip Packing Problem

Another common application of the strip packing problem is stock cutting.

- Material such as cloth, paper, or sheet metal, comes in rolls of a set width.
- These rolls may need to be cut into rectangles of arbitrary widths and heights.
- The goal is then to cut out all the required rectangles from the shortest length of roll possible, so minimising the wastage.
- the roll of material corresponds to the bin.

Strip Packing Algorithms

The problem of finding an optimal solution for the strip packing problem is NP-complete,

- Approximation algorithms find near optimal solutions but do not guarantee to find the optimal packing for every set of data.
- Most algorithms pack the rectangles into the bin using one of five approaches:
  - bottom left,
  - level-orientated,
  - split,
  - shelf or
  - hybrid.

.

Strip Packing Algorithms

Bottom-left algorithm

- rectangle to be placed as near to the bottom of the bin as it will fit
- then as far to the left as it can go at that level without overlapping any other packed rectangles.
- list of rectangles require no pre-sorting

Level-oriented algorithms

- The list of rectangles are pre-sorted into order of decreasing height.
- the packing is done on a series of levels, that the bottom of each rectangle rests on.
- The first level is the bottom of the bin and subsequent levels are defined by the height of the tallest rectangle on the previous level.

Strip Packing Algorithms

Split algorithms
The level oriented algorithms split the bin horizontally into blocks of a set width.
Split algorithms split the open ended bin vertically into smaller open ended bins depending on the widths of the rectangles.
The rectangles are first sorted by width.
Shelf algorithms
modifications of the level algorithms, that avoid pre-sorting the list of rectangles.
Rather than being determined by the tallest rectangle, the levels are fixed height shelves.
The shelf heights are set by a parameter r. ($0 < r < 1$)

There are many one-dimensional algorithms that could be used as the slave algorithm, for instance: Next-fit (NF) and First-fit (FF) algorithms.

Next-fit v's First-fit

Next-fit algorithm

- each rectangle is put onto the highest level on which it will fit, i.e. the current level of the required specifications.
- no backtracking is allowed.
- For example, if the height of the bin corresponded to time, and the time was continuously ticking by, you would not be able to go back in time and schedule any jobs for earlier points on the bin since that time has already passed!
- if the bin is a strip of material, going along a conveyor belt. The rectangles get put into position on the material as it precedes, and at the end of the conveyor they get cut out.
- In these circumstances the levels lower down the bin (closer to the cutters), are continuously getting cut up so no more rectangles could be put on these levels as they would have missed the cutting process.

© 2014 A. Tchernykh. CIMAT 2014

First-fit algorithm

- each rectangle is put onto the lowest (first) level that it will fit on.
- most economical choice since it allows to place the rectangles on all the levels lower down the bin
- suitable to use so long as all the rectangles may be arranged in the bin before anything happens to it, e.g. before the jobs are executed, or the strip starts to be cut up. Then it is fine to place the rectangles anywhere on the bin.

Strip Packing Algorithms

A. On-line v's Off-line
   - whether or not the rectangles required sorting before being placed into the bin.
   - This is an important factor to take into consideration when deciding on an appropriate algorithm for a bin packing problem.

   Consider the situation where the list of rectangles arrives one at a time.
   - When each rectangle arrives it must immediately be assigned its place in the bin.
   - Only once this has happened, does the identity of the next rectangle become known.
   - This type of environment is called on-line.
   - On-line algorithms must assume no prior knowledge of the rectangles in the list, so pre-sorting the list of rectangles is not an option.

off-line environment

- it is possible to view the whole list of rectangles first and so sort them into any order before they are placed on the bin.
- Off-line algorithms assume prior knowledge of the whole problem before any packing has to be done.

An on-line algorithm would be used in a situation where the jobs had to be done in order. For example, the items may be subject to different priorities or deadlines.

An off-line algorithm would be used when the order did not matter. For example, if pieces of material were being cut out to make a jacket, it would not matter if a sleeve was cut out before the collar, just as long as all the required pieces were produced in the end.

II. Level Algorithms and Shelf Algorithms

You can view demonstrations of all the algorithms explained below by clicking on the one you are interested in

* Next-Fit Decreasing Height

http://users.cs.cf.ac.uk/C.L.Mumford/heidi/NFDHquarters.html

* First-Fit Decreasing Height

http://users.cs.cf.ac.uk/C.L.Mumford/heidi/FFDHquarters.html

* Next-Fit Shelf

http://users.cs.cf.ac.uk/C.L.Mumford/heidi/NFSquarters.html

* First-Fit Shelf *

http://users.cs.cf.ac.uk/C.L.Mumford/heidi/FFSquarters.html

© 2014 A. Tchernykh. CIMAT 2014

The performance of all the demonstrations is assessed, and displayed on the screen after all of the rectangles have been placed in the bin. This is shown as the percentage of wasted space (space not filled by a rectangle) in the bin. It is calculated by finding the ratio of the total area of all the rectangles in the list, and the area of bin used to pack them, (the area below the top of the highest rectangle).

A. Level Algorithms
   Level algorithms are off-line algorithms.
   Their first step involves pre-sorting the list of rectangles into order of decreasing heights, i.e. the first rectangle to be packed will be the tallest and the last, the shortest.
   The packing is then made up of a series of levels.
   Each rectangle is successively packed into the bin by placing its bottom edge so as it rests on one of the levels.
   First level is the bottom of the bin and each new level is defined by drawing a horizontal line across the bin through the top of the tallest (i.e. first) rectangle on the previous level.

Next-fit decreasing height (NFDH) algorithm

The NFDH algorithm is a level algorithm which uses the next-fit approach to pack the sorted list of rectangles. The rectangles are packed, left-justified on a level until the next rectangle will not fit. This rectangle is used to define a new level and the packing continues on this level. The earlier levels are not revisited.

To view a demonstration of the NFDH algorithm click here

First-fit decreasing height (FFDH) algorithm

This is another level algorithm which this time uses the first-fit approach. Each rectangle is placed on the first (i.e. lowest) level on which it will fit. If none of the current levels have room, a new level is started.

To view a demonstration of the FFDH algorithm click here

B. Shelf Algorithms

Shelf algorithms are variants of the level algorithms which avoid pre-sorting the list of rectangles. Therefore, unlike the level algorithms, shelf algorithms are on-line. To achieve this, the levels, rather than being determined by their tallest rectangle, come in fixed sized shelves, whose heights are determined by a parameter r, ($0 < r < 1$). Each arriving rectangle is classified according to its height. Then, the packing is constructed as a series of shelves, with rectangles of similar heights being packed onto the same shelves.