

---

# APERIODIC TASK SCHEDULING

## 3.1 INTRODUCTION

This chapter presents a variety of algorithms for scheduling real-time aperiodic tasks on a single machine environment. Each algorithm represents a solution for a particular scheduling problem, which is expressed through a set of assumptions on the task set and by an optimality criterion to be used on the schedule. The restrictions made on the task set are aimed at simplifying the algorithm in terms of time complexity. When no restrictions are applied on the application tasks, the complexity can be reduced by employing heuristic approaches, which do not guarantee to find the optimal solution to a problem but can still guarantee a feasible schedule in a wide range of situations.

Although the algorithms described in this chapter are presented for scheduling aperiodic tasks on uniprocessor systems, many of them can be extended to work on multiprocessor or distributed architectures and deal with more complex task models.

To facilitate the description of the scheduling problems presented in this chapter we introduce a systematic notation that could serve as a basis for a classification scheme. Such a notation, proposed by Graham et al. [GLLK79], classifies all algorithms using three fields  $\alpha | \beta | \gamma$ , having the following meaning:

- The first field  $\alpha$  describes the machine environment on which the task set has to be scheduled (uniprocessor, multiprocessor, distributed architecture, and so on).
- The second field  $\beta$  describes task and resource characteristics (preemptive, independent versus precedence constrained, synchronous activations, and so on).

- The third field  $\gamma$  indicates the optimality criterion (performance measure) to be followed in the schedule.

For example:

- $1 \mid prec \mid L_{max}$  denotes the problem of scheduling a set of tasks with precedence constraints on a uniprocessor machine in order to minimize the maximum lateness. If no additional constraints are indicated in the second field, preemption is allowed at any time, and tasks can have arbitrary arrivals.
- $3 \mid no\_preem \mid \sum f_i$  denotes the problem of scheduling a set of tasks on a three-processor machine. Preemption is not allowed and the objective is to minimize the sum of the finishing times. Since no other constraints are indicated in the second field, tasks do not have precedence nor resource constraints but have arbitrary arrival times.
- $2 \mid sync \mid \sum Late_i$  denotes the problem of scheduling a set of tasks on a two-processor machine. Tasks have synchronous arrival times and do not have other constraints. The objective is to minimize the number of late tasks.

## 3.2 JACKSON'S ALGORITHM

The problem considered by this algorithm is  $1 \mid sync \mid L_{max}$ . That is, a set  $\mathcal{J}$  of  $n$  aperiodic tasks has to be scheduled on a single processor, minimizing the maximum lateness. All tasks consist of a single job, have synchronous arrival times, but can have different computation times and deadlines. No other constraints are considered, hence tasks must be independent; that is, cannot have precedence relations and cannot share resources in exclusive mode.

Notice that, since all tasks arrive at the same time, preemption is not an issue in this problem. In fact, preemption is effective only when tasks may arrive dynamically and newly arriving tasks have higher priority than currently executing tasks.

Without loss of generality, we assume that all tasks are activated at time  $t = 0$ , so that each job  $J_i$  can be completely characterized by two parameters: a computation time  $C_i$  and a relative deadline  $D_i$  (which, in this case, is also equal to the absolute deadline). Thus, the task set  $\mathcal{J}$  can be denoted as

$$\mathcal{J} = \{J_i(C_i, D_i), i = 1, \dots, n\}.$$

A simple algorithm that solves this problem was found by Jackson in 1955. It is called *Earliest Due Date* (EDD) and can be expressed by the following rule [Jac55]:

**Theorem 3.1 (Jackson's rule)** *Given a set of  $n$  independent tasks, any algorithm that executes the tasks in order of nondecreasing deadlines is optimal with respect to minimizing the maximum lateness.*

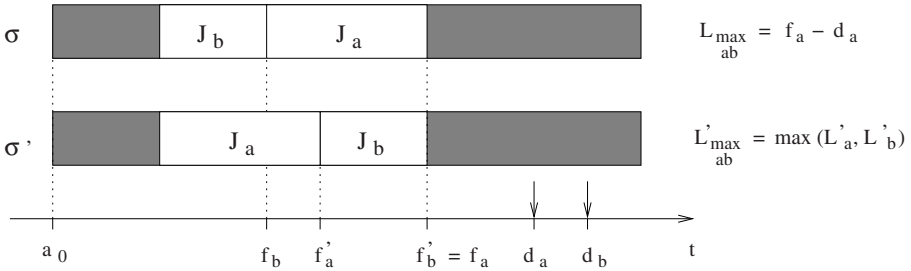
**Proof.** Jackson's theorem can be proved by a simple interchange argument. Let  $\sigma$  be a schedule produced by any algorithm  $A$ . If  $A$  is different than EDD, then there exist two tasks  $J_a$  and  $J_b$ , with  $d_a \leq d_b$ , such that  $J_b$  immediately precedes  $J_a$  in  $\sigma$ . Now, let  $\sigma'$  be a schedule obtained from  $\sigma$  by exchanging  $J_a$  with  $J_b$ , so that  $J_a$  immediately precedes  $J_b$  in  $\sigma'$ .

As illustrated in Figure 3.1, interchanging the position of  $J_a$  and  $J_b$  in  $\sigma$  cannot increase the maximum lateness. In fact, the maximum lateness between  $J_a$  and  $J_b$  in  $\sigma$  is  $L_{max}(a, b) = f_a - d_a$ , whereas the maximum lateness between  $J_a$  and  $J_b$  in  $\sigma'$  can be written as  $L'_{max}(a, b) = max(L'_a, L'_b)$ . Two cases must be considered:

1. If  $L'_a \geq L'_b$ , then  $L'_{max}(a, b) = f'_a - d_a$ , and, since  $f'_a < f_a$ , we have  $L'_{max}(a, b) < L_{max}(a, b)$ .
2. If  $L'_a \leq L'_b$ , then  $L'_{max}(a, b) = f'_b - d_b = f_a - d_b$ , and, since  $d_a < d_b$ , we have  $L'_{max}(a, b) < L_{max}(a, b)$ .

Since, in both cases,  $L'_{max}(a, b) < L_{max}(a, b)$ , we can conclude that interchanging  $J_a$  and  $J_b$  in  $\sigma$  cannot increase the maximum lateness of the task set. By a finite number of such transpositions,  $\sigma$  can be transformed in  $\sigma_{EDD}$  and, since in each transposition the maximum lateness cannot increase,  $\sigma_{EDD}$  is optimal.  $\square$

The complexity required by Jackson's algorithm to build the optimal schedule is due to the procedure that sorts the tasks by increasing deadlines. Hence, if the task set consists of  $n$  tasks, the complexity of the EDD algorithm is  $O(n \log n)$ .



if  $(L'_a \geq L'_b)$  then  $L'_{ab}{}^{\max} = f'_a - d_a < f_a - d_a$

if  $(L'_a \leq L'_b)$  then  $L'_{ab}{}^{\max} = f'_b - d_b < f_a - d_a$

in both cases:  $L'_{ab}{}^{\max} < L_{ab}^{\max}$

Figure 3.1 Optimality of Jackson’s algorithm.

### 3.2.1 EXAMPLES

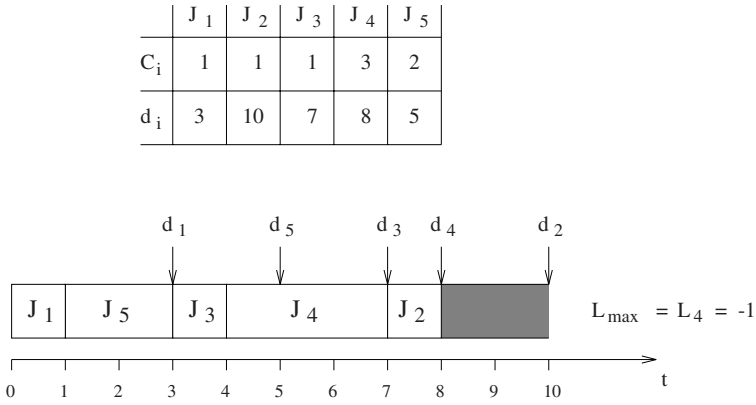
#### EXAMPLE 1

Consider a set of five tasks, simultaneously activated at time  $t = 0$ , whose parameters (worst-case computation times and deadlines) are indicated in the table shown in Figure 3.2. The schedule of the tasks produced by the EDD algorithm is also depicted in Figure 3.2. The maximum lateness is equal to  $-1$  and it is due to task  $J_4$ , which completes a unit of time before its deadline. Since the maximum lateness is negative, we can conclude that all tasks have been executed within their deadlines.

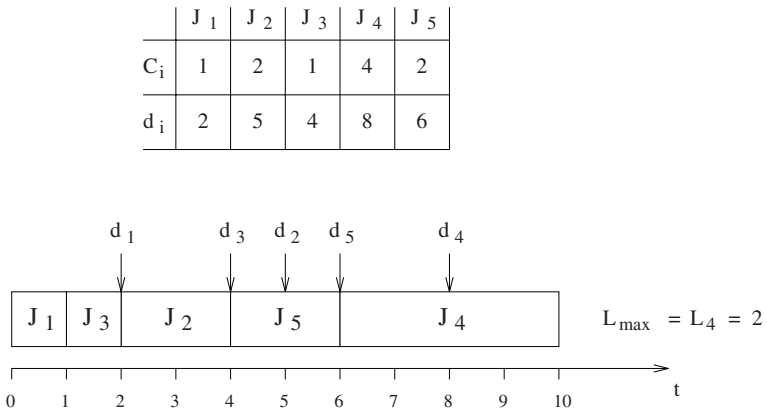
Note that the optimality of the EDD algorithm cannot guarantee the feasibility of the schedule for any task set. It only guarantees that if a feasible schedule exists for a task set, then EDD will find it.

#### EXAMPLE 2

Figure 3.3 illustrates an example in which the task set cannot be feasibly scheduled. Still, however, EDD produces the optimal schedule that minimizes the maximum lateness. Note that since  $J_4$  misses its deadline, the maximum lateness is greater than zero ( $L_{max} = L_4 = 2$ ).



**Figure 3.2** A feasible schedule produced by Jackson's algorithm.



**Figure 3.3** An infeasible schedule produced by Jackson's algorithm.

### 3.2.2 GUARANTEE

To guarantee that a set of tasks can be feasibly scheduled by the EDD algorithm, we need to show that, in the worst case, all tasks can complete before their deadlines. This means that we have to show that for each task, the worst-case finishing time  $f_i$  is less than or equal to its deadline  $d_i$ :

$$\forall i = 1, \dots, n \quad f_i \leq d_i.$$

If tasks have hard timing requirements, such a schedulability analysis must be done before actual tasks' execution. Without loss of generality, we can assume that tasks  $J_1, J_2, \dots, J_n$  are listed by increasing deadlines, so that  $J_1$  is the task with the earliest deadline. In this case, the worst-case finishing time of task  $J_i$  can be easily computed as

$$f_i = \sum_{k=1}^i C_k.$$

Therefore, if the task set consists of  $n$  tasks, the guarantee test can be performed by verifying the following  $n$  conditions:

$$\forall i = 1, \dots, n \quad \sum_{k=1}^i C_k \leq d_i. \quad (3.1)$$

## 3.3 HORN'S ALGORITHM

If tasks are not synchronous but can have arbitrary arrival times (that is, tasks can be activated dynamically during execution), then preemption becomes an important factor. In general, a scheduling problem in which preemption is allowed is always easier than its non-preemptive counterpart. In a non-preemptive scheduling algorithm, the scheduler must ensure that a newly arriving task will never need to interrupt a currently executing task in order to meet its own deadline. This guarantee requires a considerable amount of searching. If preemption is allowed, however, this searching is unnecessary, since a task can be interrupted if a more important task arrives [WR91].

In 1974, Horn found an elegant solution to the problem of scheduling a set of  $n$  independent tasks on a uniprocessor system, when tasks may have dynamic arrivals and preemption is allowed ( $1 \mid \text{preem} \mid L_{max}$ ).

The algorithm, called *Earliest Deadline First* (EDF), can be expressed by the following theorem [Hor74]:

**Theorem 3.2 (Horn)** *Given a set of  $n$  independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among all the ready tasks is optimal with respect to minimizing the maximum lateness.*

This result can be proved by an interchange argument similar to the one used by Jackson. The formal proof of the EDF optimality was given by Dertouzos in 1974 [Der74] and it is illustrated below. The complexity of the algorithm is  $O(n)$  per task, if the ready queue is implemented as a list, or  $O(n \log n)$  per task, if the ready queue is implemented as a heap.

### 3.3.1 EDF OPTIMALITY

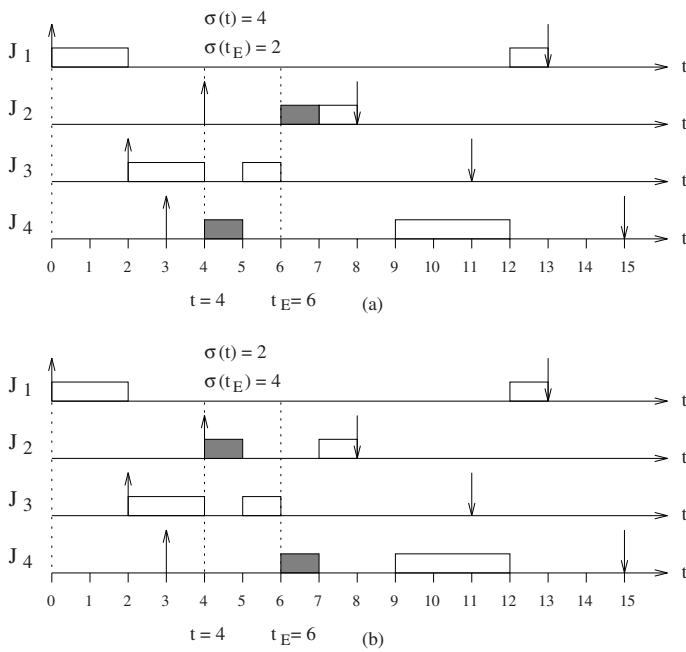
The original proof provided by Dertouzos [Der74] shows that EDF is optimal in the sense of feasibility. This means that if there exists a feasible schedule for a task set  $\mathcal{J}$ , then EDF is able to find it. The proof can easily be extended to show that EDF also minimizes the maximum lateness. This is more general because an algorithm that minimizes the maximum lateness is also optimal in the sense of feasibility. The contrary is not true.

Using the same approach proposed by Dertouzos, let  $\sigma$  be the schedule produced by a generic algorithm  $A$  and let  $\sigma_{EDF}$  be the schedule obtained by the EDF algorithm. Since preemption is allowed, each task can be executed in disjointed time intervals. Without loss of generality, the schedule  $\sigma$  can be divided into *time slices* of one unit of time each. To simplify the formulation of the proof, let us define the following abbreviations:

- $\sigma(t)$  identifies the task executing in the slice  $[t, t + 1)$ .<sup>1</sup>
- $E(t)$  identifies the ready task that, at time  $t$ , has the earliest deadline.
- $t_E(t)$  is the time ( $\geq t$ ) at which the next slice of task  $E(t)$  begins its execution in the current schedule.

If  $\sigma \neq \sigma_{EDF}$ , then in  $\sigma$  there exists a time  $t$  such that  $\sigma(t) \neq E(t)$ . As illustrated in Figure 3.4, the basic idea used in the proof is that interchanging the position of  $\sigma(t)$  and  $E(t)$  cannot increase the maximum lateness. If the schedule  $\sigma$  starts at time  $t = 0$  and  $D$  is the latest deadline of the task set ( $D = \max_i \{d_i\}$ ) then  $\sigma_{EDF}$  can be obtained from  $\sigma$  by at most  $D$  transpositions.

<sup>1</sup> $[a,b)$  denotes an interval of values  $x$  such that  $a \leq x < b$ .



**Figure 3.4** Proof of the optimality of the EDF algorithm. **a.** schedule  $\sigma$  at time  $t = 4$ . **b.** new schedule obtained after a transposition.



```

Algorithm: interchange
{
  for (t=0 to D-1) {
    if ( $\sigma(t) \neq E(t)$ ) {
       $\sigma(t_E) = \sigma(t)$ ;
       $\sigma(t) = E(t)$ ;
    }
  }
}

```

**Figure 3.5** Transformation algorithm used by Dertouzos to prove the optimality of EDF.

The algorithm used by Dertouzos to transform any schedule  $\sigma$  into an EDF schedule is illustrated in Figure 3.5. For each time slice  $t$ , the algorithm verifies whether the task  $\sigma(t)$  scheduled in the slice  $t$  is the one with the earliest deadline,  $E(t)$ . If it is, nothing is done, otherwise a transposition takes place and the slices at  $t$  and  $t_E$  are exchanged (see Figure 3.4). In particular, the slice of task  $E(t)$  is anticipated at time  $t$ , while the slice of task  $\sigma(t)$  is postponed at time  $t_E$ . Using the same argument adopted in the proof of Jackson's theorem, it is easy to show that after each transposition the maximum lateness cannot increase; therefore, EDF is optimal.

By applying the interchange algorithm to the schedule shown in Figure 3.4a, the first transposition occurs at time  $t = 4$ . At this time, in fact, the CPU is assigned to  $J_4$ , but the task with the earliest deadline is  $J_2$ , which is scheduled at time  $t_E = 6$ . As a consequence, the two slices in gray are exchanged and the resulting schedule is shown in Figure 3.4b. The algorithm examines all slices, until  $t = D$ , performing a slice exchange when necessary.

To show that a transposition preserves the schedulability, note that at any instant each slice in  $\sigma$  can be either anticipated or postponed up to  $t_E$ . If a slice is anticipated, the feasibility of that task is obviously preserved. If a slice of  $J_i$  is postponed at  $t_E$  and  $\sigma$  is feasible, it must be  $(t_E + 1) \leq d_E$ , being  $d_E$  the earliest deadline. Since  $d_E \leq d_i$  for any  $i$ , then we have  $t_E + 1 \leq d_i$ , which guarantees the schedulability of the slice postponed at  $t_E$ .

### 3.3.2 EXAMPLE

An example of schedule produced by the EDF algorithm on a set of five tasks is shown in Figure 3.6. At time  $t = 0$ , tasks  $J_1$  and  $J_2$  arrive and, since  $d_1 < d_2$ , the processor is assigned to  $J_1$ , which completes at time  $t = 1$ . At time  $t = 2$ , when  $J_2$  is executing, task  $J_3$  arrives and preempts  $J_2$ , being  $d_3 < d_2$ . Note that, at time  $t = 3$ , the arrival of  $J_4$  does not interrupt the execution of  $J_3$ , because  $d_3 < d_4$ . As  $J_3$  is completed, the processor is assigned to  $J_2$ , which resumes and executes until completion. Then  $J_4$  starts at  $t = 5$ , but, at time  $t = 6$ , it is preempted by  $J_5$ , which has an earlier deadline. Task  $J_4$  resumes at time  $t = 8$ , when  $J_5$  is completed. Notice that all tasks meet their deadlines and the maximum lateness is  $L_{max} = L_2 = 0$ .

	$J_1$	$J_2$	$J_3$	$J_4$	$J_5$
$a_i$	0	0	2	3	6
$C_i$	1	2	2	2	2
$d_i$	2	5	4	10	9

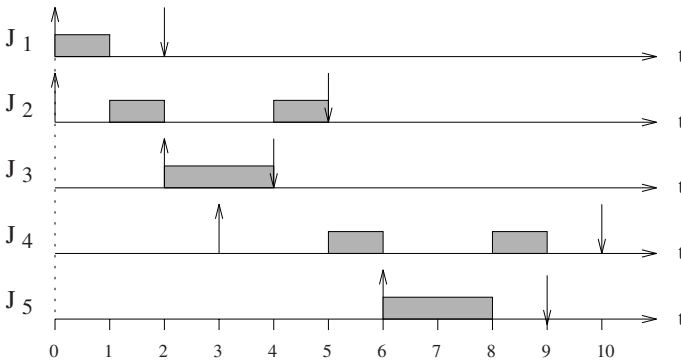


Figure 3.6 Example of EDF schedule.

### 3.3.3 GUARANTEE

When tasks have dynamic activations and the arrival times are not known a priori, the guarantee test has to be done dynamically, whenever a new task enters the system. Let  $\mathcal{J}$  be the current set of active tasks, which have been previously guaranteed, and

let  $J_{new}$  be a newly arrived task. In order to accept  $J_{new}$  in the system we have to guarantee that the new task set  $\mathcal{J}' = \mathcal{J} \cup \{J_{new}\}$  is also schedulable.

Following the same approach used in EDD, to guarantee that the set  $\mathcal{J}'$  is feasibly schedulable by EDF, we need to show that, in the worst case, all tasks in  $\mathcal{J}'$  will complete before their deadlines. This means that we have to show that, for each task, the worst-case finishing time  $f_i$  is less than or equal to its deadline  $d_i$ .

Without loss of generality, we can assume that all tasks in  $\mathcal{J}'$  (including  $J_{new}$ ) are ordered by increasing deadlines, so that  $J_1$  is the task with the earliest deadline. Moreover, since tasks are preemptable, when  $J_{new}$  arrives at time  $t$  some tasks could have been partially executed. Thus, let  $c_i(t)$  be the remaining worst-case execution time of task  $J_i$  (notice that  $c_i(t)$  has an initial value equal to  $C_i$  and can be updated whenever  $J_i$  is preempted). Hence, at time  $t$ , the worst-case finishing time of task  $J_i$  can be easily computed as

$$f_i = \sum_{k=1}^i c_k(t).$$

Thus, the schedulability can be guaranteed by the following conditions:

$$\forall i = 1, \dots, n \quad \sum_{k=1}^i c_k(t) \leq d_i. \quad (3.2)$$

Noting that  $f_i = f_{i-1} + c_i(t)$  (where  $f_0 = 0$  by definition), the dynamic guarantee test can be performed in  $O(n)$  by executing the algorithm shown in Figure 3.7.

### 3.4 NON-PREEMPTIVE SCHEDULING

When preemption is not allowed and tasks can have arbitrary arrivals, the problem of minimizing the maximum lateness and the problem of finding a feasible schedule become NP-hard [GLLK79, LRKB77, KIM78]. Figure 3.8 illustrates an example that shows that EDF is no longer optimal if tasks cannot be preempted during their execution. In fact, although a feasible schedule exists for that task set (see Figure 3.8a), EDF does not produce a feasible schedule (see Figure 3.8b), since  $J_2$  executes one unit of time after its deadline. This happens because EDF immediately assigns the processor to task  $J_1$ ; thus, when  $J_2$  arrives at time  $t = 1$ ,  $J_1$  cannot be preempted.  $J_2$  can start only at time  $t = 4$ , after  $J_1$  completion, but it is too late to meet its deadline.

Notice, however, that in the optimal schedule shown in Figure 3.8a the processor remains idle in the interval  $[0, 1)$  although  $J_1$  is ready to execute. If arrival times are not

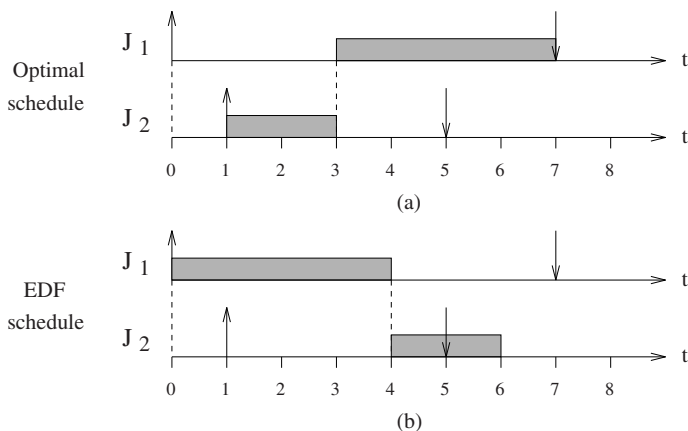
```

Algorithm: EDF_guarantee( $\mathcal{J}, J_{new}$ )
{
   $\mathcal{J}' = \mathcal{J} \cup \{J_{new}\};$     /* ordered by deadline */
   $t = \text{current\_time}();$ 
   $f_0 = 0;$ 
  for (each  $J_i \in \mathcal{J}'$ ) {
     $f_i = f_{i-1} + c_i(t);$ 
    if ( $f_i > d_i$ ) return(UNFEASIBLE);
  }
  return(FEASIBLE);
}

```

**Figure 3.7** EDF guarantee algorithm.

	$J_1$	$J_2$
$a_i$	0	1
$C_i$	4	2
$d_i$	7	5



**Figure 3.8** EDF is not optimal in a non-preemptive model. In fact, although there exists a feasible schedule (a), the schedule produced by EDF (b) is infeasible.



ing paths on the tree and build a complete schedule in polynomial time. Heuristic algorithms may produce a feasible schedule in polynomial time; however, they do not guarantee to find it, since they do not explore all possible solutions.

### 3.4.1 BRATLEY'S ALGORITHM

(1 | *NO-PREEM* | *FEASIBLE*)

The following algorithm was proposed by Bratley et al. in 1971 [BFR71] to solve the problem of finding a feasible schedule of a set of non-preemptive tasks with arbitrary arrival times. The algorithm starts with an empty schedule and, at each step of the search, visits a new vertex and adds a task in the partial schedule. With respect to the exhaustive search, Bratley's algorithm uses a pruning technique to determine when a current search can be reasonably abandoned. In particular, a branch is abandoned when

- the addition of any node to the current path causes a missed deadline;
- a feasible schedule is found at the current path.

To better understand the pruning technique adopted by the algorithm, consider the task set shown in Figure 3.10, which also illustrates the paths analyzed in the tree space.

To follow the evolution of the algorithm, the numbers inside each node of the tree indicate which task is being scheduled in that path, whereas the numbers beside the nodes represent the time at which the indicated task completes its execution. Whenever the addition of any node to the current path causes a missed deadline, the corresponding branch is abandoned and the task causing the timing fault is labeled with a (†).

In the example, the first task considered for extending the empty schedule is  $J_1$ , whose index is written in the first node of the leftmost branch of the tree. Since  $J_1$  arrives at  $t = 4$  and requires two units of processing time, its worst-case finishing time is  $f_1 = 6$ , indicated beside the correspondent node. Before expanding the branch, however, the pruning mechanism checks whether the addition of any node to the current path may cause a timing fault, and it discovers that task  $J_2$  would miss its deadline, if added. As a consequence, the search on this branch is abandoned and a considerable amount of computation is avoided.

In the average case, pruning techniques are very effective for reducing the search space. Nevertheless, the worst-case complexity of the algorithm is still  $O(n \cdot n!)$ .

	$J_1$	$J_2$	$J_3$	$J_4$
$a_i$	4	1	1	0
$C_i$	2	1	2	2
$d_i$	7	5	6	4

Number in the node = scheduled task

Number outside the node = finishing time

$J_i^+$  = task that misses its deadline

⊙ = feasible schedule

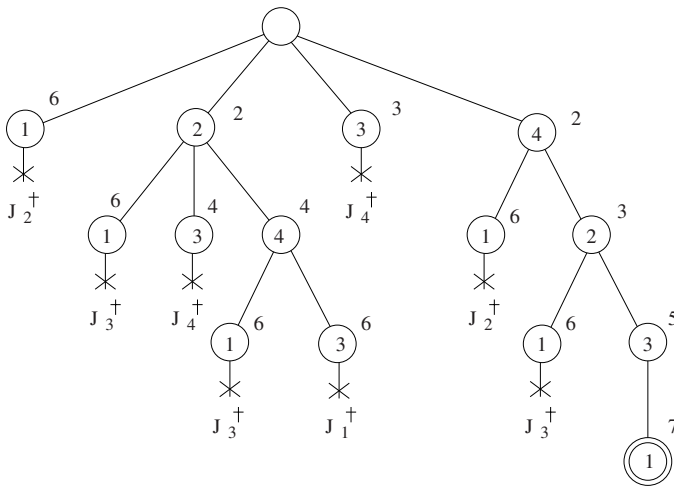


Figure 3.10 Example of search performed by Bratley's algorithm.

For this reason, Bratley's algorithm can only be used in off-line mode, when all task parameters (including the arrival times) are known in advance. This can be the case of a time-triggered system, where tasks are activated at predefined instants by a timer process.

As in most off-line real-time systems, the resulting schedule produced by Bratley's algorithm can be stored in a data structure, called *task activation list*. Then, at run time, a dispatcher simply extracts the next task from the activation list and puts it in execution.

### 3.4.2 THE SPRING ALGORITHM

Here we describe the scheduling algorithm adopted in the *Spring* kernel [SR87, SR91], a hard real-time kernel designed at the University of Massachusetts at Amherst by Stankovic and Ramamritham to support critical control applications in dynamic environments. The objective of the algorithm is to find a feasible schedule when tasks have different types of constraints, such as precedence relations, resource constraints, arbitrary arrivals, non-preemptive properties, and importance levels. The Spring algorithm is used in a distributed computer architecture and can also be extended to include fault-tolerance requirements.

Clearly, this problem is *NP*-hard and finding a feasible schedule would be too expensive in terms of computation time, especially for dynamic systems. In order to make the algorithm computationally tractable even in the worst case, the search is driven by a *heuristic function*  $H$ , which actively directs the scheduling to a plausible path. On each level of the search, function  $H$  is applied to each of the tasks that remain to be scheduled. The task with the smallest value determined by the heuristic function  $H$  is selected to extend the current schedule.

The heuristic function is a very flexible mechanism to easily define and modify the scheduling policy of the kernel. For example, if  $H = a_i$  (arrival time) the algorithm behaves as First Come First Served, if  $H = C_i$  (computation time) it works as Shortest Job First, whereas if  $H = d_i$  (deadline) the algorithm is equivalent to Earliest Deadline First.

To consider resource constraints in the scheduling algorithm, each task  $J_i$  has to declare a binary array of resources  $R_i = [R_1(i), \dots, R_r(i)]$ , where  $R_k(i) = 0$  if  $J_i$  does not use resource  $R_k$ , and  $R_k(i) = 1$  if  $J_i$  uses  $R_k$  in exclusive mode. Given a partial schedule, the algorithm determines, for each resource  $R_k$ , the earliest time the resource is available. This time is denoted as  $EAT_k$  (Earliest Available Time). Thus, the earliest start time that a task  $J_i$  can begin the execution without blocking on shared resources is

$$T_{est}(i) = \max[a_i, \max_k(EAT_k)],$$

where  $a_i$  is the arrival time of  $J_i$ . Once  $T_{est}$  is calculated for all the tasks, a possible search strategy is to select the task with the smallest value of  $T_{est}$ . Composed heuristic functions can also be used to integrate relevant information on the tasks, such as

$$\begin{aligned} H &= d + W \cdot C \\ H &= d + W \cdot T_{est}, \end{aligned}$$



$H = a$	First Come First Served (FCFS)
$H = C$	Shortest Job First (SJF)
$H = d$	Earliest Deadline First (EDF)
$H = T_{\text{est}}$	Earliest Start Time First (ESTF)
$H = d + w C$	EDF + SJF
$H = d + w T_{\text{est}}$	EDF + ESTF

**Figure 3.11** Example of heuristic functions that can be adopted in the Spring algorithm.

where  $W$  is a weight that can be adjusted for different application environments. Figure 3.11 shows some possible heuristic functions that can be used in Spring to direct the search process.

In order to handle precedence constraints, another factor  $E$ , called *eligibility*, is added to the heuristic function. A task becomes eligible to execute ( $E_i = 1$ ) only when all its ancestors in the precedence graph are completed. If a task is not eligible, then  $E_i = \infty$ ; hence, it cannot be selected for extending a partial schedule.

While extending a partial schedule, the algorithm determines whether the current schedule is *strongly feasible*; that is, also feasible by extending it with any of the remaining tasks. If a partial schedule is found not to be strongly feasible, the algorithm stops the search process and announces that the task set is not schedulable, otherwise the search continues until a complete feasible schedule is met. Since a feasible schedule is reached through  $n$  nodes and each partial schedule requires the evaluation of most  $n$  heuristic functions, the complexity of the Spring algorithm is  $O(n^2)$ .

Backtracking can be used to continue the search after a failure. In this case, the algorithm returns to the previous partial schedule and extends it by the task with the second smallest heuristic value. To restrict the overhead of backtracking, however, the maximum number of possible backtracks must be limited. Another method to reduce the complexity is to restrict the number of evaluations of the heuristic function. Due to that, if a partial schedule is found to be strongly feasible, the heuristic function is applied not to all the remaining tasks but only to the  $k$  remaining tasks with the earliest deadlines. Given that only  $k$  tasks are considered at each step, the complexity becomes

$O(kn)$ . If the value of  $k$  is constant (and small, compared to the task set size), then the complexity becomes linearly proportional to the number of tasks.

A disadvantage of the heuristic scheduling approach is that it is not optimal. This means that, if there is a feasible schedule, the Spring algorithm may not find it.

## 3.5 SCHEDULING WITH PRECEDENCE CONSTRAINTS

The problem of finding an optimal schedule for a set of tasks with precedence relations is in general  $NP$ -hard. However, optimal algorithms that solve the problem in polynomial time can be found under particular assumptions on the tasks. In this section we present two algorithms that minimize the maximum lateness by assuming synchronous activations and preemptive scheduling, respectively.

### 3.5.1 LATEST DEADLINE FIRST

(1 | *PREC*, *SYNC* |  $L_{MAX}$ )

In 1973, Lawler [Law73] presented an optimal algorithm that minimizes the maximum lateness of a set of tasks with precedence relations and simultaneous arrival times. The algorithm is called *Latest Deadline First* (LDF) and can be executed in polynomial time with respect to the number of tasks in the set.

Given a set  $\mathcal{J}$  of  $n$  tasks and a directed acyclic graph (DAG) describing their precedence relations, LDF builds the scheduling queue from tail to head: among the tasks without successors or whose successors have been all selected, LDF selects the task with the latest deadline to be scheduled last. This procedure is repeated until all tasks in the set are selected. At run time, tasks are extracted from the head of the queue, so that the first task inserted in the queue will be executed last, whereas the last task inserted in the queue will be executed first.

The correctness of this rule is proved as follows: Let  $\mathcal{J}$  be the complete set of tasks to be scheduled, let  $\Gamma \subseteq \mathcal{J}$  be the subset of tasks without successors, and let  $J_l$  be the task in  $\Gamma$  with the latest deadline  $d_l$ . If  $\sigma$  is any schedule that does not follow the EDL rule, then the last scheduled task, say  $J_k$ , will not be the one with the latest deadline; thus  $d_k \leq d_l$ . Since  $J_l$  is scheduled before  $J_k$ , let us partition  $\Gamma$  into four subsets, so

that  $\Gamma = A \cup \{J_l\} \cup B \cup \{J_k\}$ . Clearly, in  $\sigma$  the maximum lateness for  $\Gamma$  is greater or equal to  $L_k = f - d_k$ , where  $f = \sum_{i=1}^n C_i$  is the finishing time of task  $J_k$ .

We show that moving  $J_l$  to the end of the schedule cannot increase the maximum lateness in  $\Gamma$ , which proves the optimality of LDF. To do that, let  $\sigma^*$  be the schedule obtained from  $\sigma$  after moving task  $J_l$  to the end of the queue and shifting all other tasks to the left. The two schedules  $\sigma$  and  $\sigma^*$  are depicted in Figure 3.12. Clearly, in  $\sigma^*$  the maximum lateness for  $\Gamma$  is given by

$$L_{max}^*(\Gamma) = \max[L_{max}^*(A), L_{max}^*(B), L_k^*, L_l^*].$$

Each argument of the max function is no greater than  $L_{max}(\Gamma)$ . In fact,

- $L_{max}^*(A) = L_{max}(A) \leq L_{max}(\Gamma)$  because  $A$  is not moved;
- $L_{max}^*(B) \leq L_{max}(B) \leq L_{max}(\Gamma)$  because  $B$  starts earlier in  $\sigma^*$ ;
- $L_k^* \leq L_k \leq L_{max}(\Gamma)$  because task  $J_k$  starts earlier in  $\sigma^*$ ;
- $L_l^* = f - d_l \leq f - d_k \leq L_{max}(\Gamma)$  because  $d_k \leq d_l$ .

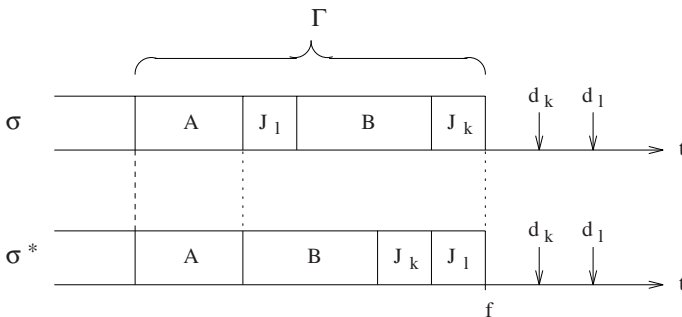
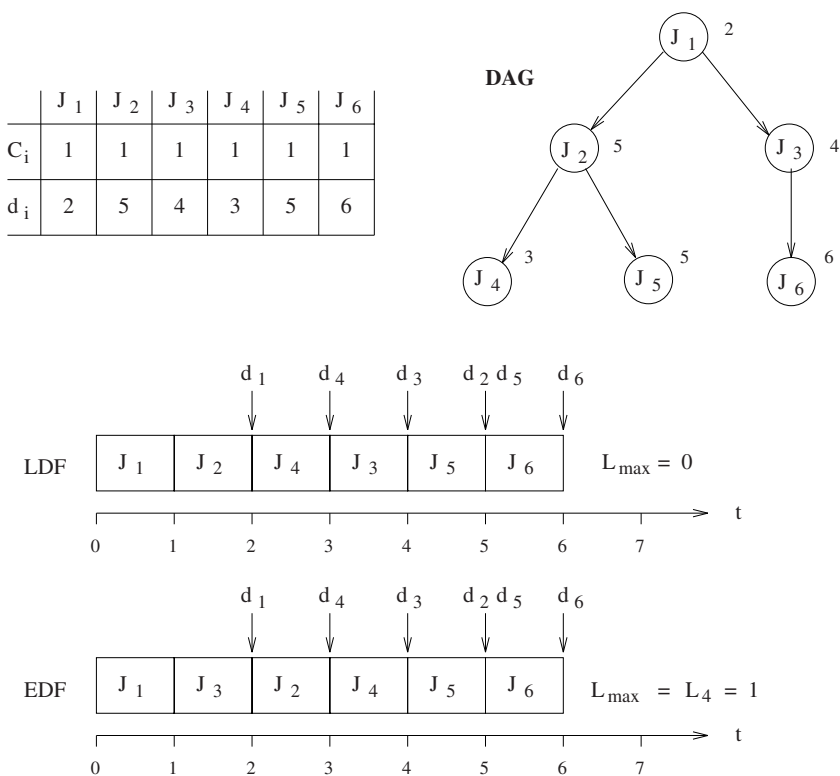


Figure 3.12 Proof of LDF optimality.

Since  $L_{max}^*(\Gamma) \leq L_{max}(\Gamma)$ , moving  $J_l$  to the end of the schedule does not increase the maximum lateness in  $\Gamma$ . This means that scheduling last the task  $J_l$  with the latest deadline minimizes the maximum lateness in  $\Gamma$ . Then, removing this task from  $\mathcal{J}$  and repeating the argument for the remaining  $n - 1$  tasks in the set  $\mathcal{J} - \{J_l\}$ , LDF can find the second-to-last task in the schedule, and so on. The complexity of the LDF algorithm is  $O(n^2)$ , since for each of the  $n$  steps it needs to visit the precedence graph to find the subset  $\Gamma$  with no successors.

Consider the example depicted in Figure 3.13, which shows the parameters of six tasks together with their precedence graph. The numbers beside each node of the graph indicate task deadlines. Figure 3.13 also shows the schedule produced by EDF to highlight the differences between the two approaches. The EDF schedule is constructed by selecting the task with the earliest deadline among the current eligible tasks. Notice that EDF is not optimal under precedence constraints, since it achieves a greater  $L_{max}$  with respect to LDF.



**Figure 3.13** Comparison between schedules produced by LDF and EDF on a set of tasks with precedence constraints.

### 3.5.2 EDF WITH PRECEDENCE CONSTRAINTS

(1 | *PREC, PREEM* | *L<sub>MAX</sub>*)

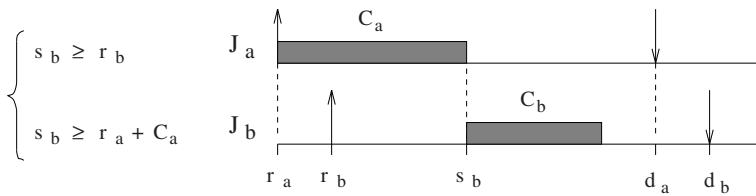
The problem of scheduling a set of  $n$  tasks with precedence constraints and dynamic activations can be solved in polynomial time complexity only if tasks are preemptable. In 1990, Chetto, Silly, and Bouchentouf [CSB90] presented an algorithm that solves this problem in elegant fashion. The basic idea of their approach is to transform a set  $\mathcal{J}$  of dependent tasks into a set  $\mathcal{J}^*$  of independent tasks by an adequate modification of timing parameters. Then, tasks are scheduled by the Earliest Deadline First (EDF) algorithm. The transformation algorithm ensures that  $\mathcal{J}$  is schedulable and the precedence constraints are obeyed if and only if  $\mathcal{J}^*$  is schedulable. Basically, all release times and deadlines are modified so that each task cannot start before its predecessors and cannot preempt their successors.

#### MODIFICATION OF THE RELEASE TIMES

The rule for modifying tasks' release times is based on the following observation. Given two tasks  $J_a$  and  $J_b$ , such that  $J_a \rightarrow J_b$  (that is,  $J_a$  is an immediate predecessor of  $J_b$ ), then in any valid schedule that meets precedence constraints the following conditions must be satisfied (see Figure 3.14):

$s_b \geq r_b$  (that is,  $J_b$  must start the execution not earlier than its release time);

$s_b \geq r_a + C_a$  (that is,  $J_b$  must start the execution not earlier than the minimum finishing time of  $J_a$ ).



**Figure 3.14** If  $J_a \rightarrow J_b$ , then the release time of  $J_b$  can be replaced by  $\max(r_b, r_a + C_a)$ .

Therefore, the release time  $r_b$  of  $J_b$  can be replaced by the maximum between  $r_b$  and  $(r_a + C_a)$  without changing the problem. Let  $r_b^*$  be the new release time of  $J_b$ . Then,

$$r_b^* = \max(r_b, r_a + C_a).$$

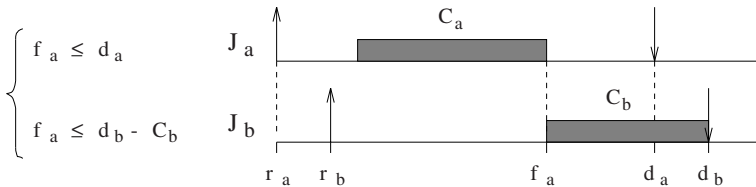
The algorithm that modifies the release times can be implemented in  $O(n^2)$  and can be described as follows:

1. For any initial node of the precedence graph, set  $r_i^* = r_i$ .
2. Select a task  $J_i$  such that its release time has not been modified but the release times of all immediate predecessors  $J_h$  have been modified. If no such task exists, exit.
3. Set  $r_i^* = \max[r_i, \max(r_h^* + C_h : J_h \rightarrow J_i)]$ .
4. Return to step 2.

### MODIFICATION OF THE DEADLINES

The rule for modifying tasks' deadlines is based on the following observation. Given two tasks  $J_a$  and  $J_b$ , such that  $J_a \rightarrow J_b$  (that is,  $J_a$  is an immediate predecessor of  $J_b$ ), then in any feasible schedule that meets the precedence constraints the following conditions must be satisfied (see Figure 3.15):

- $f_a \leq d_a$  (that is,  $J_a$  must finish the execution within its deadline);
- $f_a \leq d_b - C_b$  (that is,  $J_a$  must finish the execution not later than the maximum start time of  $J_b$ ).



**Figure 3.15** If  $J_a \rightarrow J_b$ , then the deadline of  $J_a$  can be replaced by  $\min(d_a, d_b - C_b)$ .

Therefore, the deadline  $d_a$  of  $J_a$  can be replaced by the minimum between  $d_a$  and  $(d_b - C_b)$  without changing the problem. Let  $d_a^*$  be the new deadline of  $J_a$ . Then,

$$d_a^* = \min(d_a, d_b - C_b).$$

The algorithm that modifies the deadlines can be implemented in  $O(n^2)$  and can be described as follows:

1. For any terminal node of the precedence graph, set  $d_i^* = d_i$ .
2. Select a task  $J_i$  such that its deadline has not been modified but the deadlines of all immediate successors  $J_k$  have been modified. If no such task exists, exit.
3. Set  $d_i^* = \min[d_i, \min(d_k^* - C_k : J_i \rightarrow J_k)]$ .
4. Return to step 2.

## PROOF OF OPTIMALITY

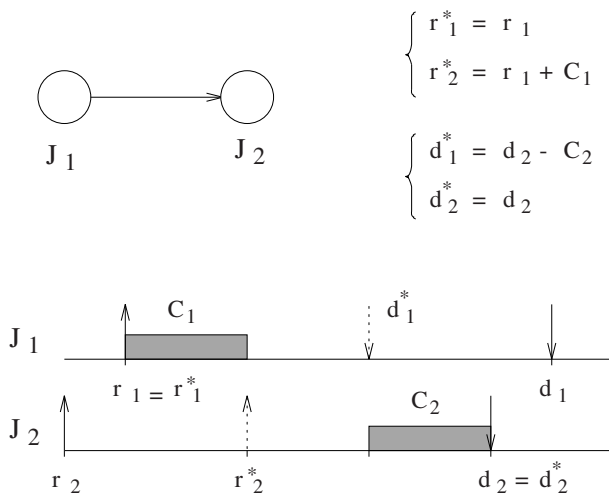
The transformation algorithm ensures that if a feasible schedule exists for the modified task set  $\mathcal{J}^*$  under EDF, then the original task set  $\mathcal{J}$  is also schedulable; that is, all tasks in  $\mathcal{J}$  meet both precedence and timing constraints. In fact, if  $\mathcal{J}^*$  is schedulable, all modified tasks start at or after time  $r_i^*$  and are completed at or before time  $d_i^*$ . Since  $r_i^* \geq r_i$  and  $d_i^* \leq d_i$ , the schedulability of  $\mathcal{J}^*$  implies that  $\mathcal{J}$  is also schedulable.

To show that precedence relations in  $\mathcal{J}$  are not violated, consider the example illustrated in Figure 3.16, where  $J_1$  must precede  $J_2$  (i.e.,  $J_1 \rightarrow J_2$ ), but  $J_2$  arrives before  $J_1$  and has an earlier deadline. Clearly, if the two tasks are executed under EDF, their precedence relation cannot be met. However, if we apply the transformation algorithm, the time constraints are modified as follows:

$$\begin{cases} r_1^* = r_1 \\ r_2^* = \max(r_2, r_1 + C_1) \end{cases} \quad \begin{cases} d_1^* = \min(d_1, d_2 - C_2) \\ d_2^* = d_2 \end{cases}$$

This means that, since  $r_2^* > r_1^*$ ,  $J_2$  cannot start before  $J_1$ . Moreover,  $J_2$  cannot preempt  $J_1$  because  $d_1^* < d_2^*$  and, based on EDF, the processor is assigned to the task with the earliest deadline. Hence, the precedence relation is respected.

In general, for any pair of tasks such that  $J_i \prec J_j$ , we have  $r_i^* \leq r_j^*$  and  $d_i^* \leq d_j^*$ . This means that, if  $J_i$  is in execution, then all successors of  $J_i$  cannot start before  $r_i$  because



**Figure 3.16** The transformation algorithm preserves the timing and the precedence constraints.

$r_i^* \leq r_j^*$ . Moreover, they cannot preempt  $J_i$  because  $d_i^* \leq d_j^*$  and, according to EDF, the processor is assigned to the ready task having the earliest deadline. Therefore, both timing and precedence constraints specified for task set  $\mathcal{J}$  are guaranteed by the schedulability of the modified set  $\mathcal{J}^*$ .

### 3.6 SUMMARY

The scheduling algorithms described in this chapter for handling real-time tasks with aperiodic arrivals can be compared in terms of assumptions on the task set and computational complexity. Figure 3.17 summarizes the main characteristics of such algorithms and can be used for selecting the most appropriate scheduling policy for a particular problem.



	sync. activation	preemptive async. activation	non-preemptive async. activation
independent	<b>EDD</b> (Jackson '55) $O(n \log n)$ Optimal	<b>EDF</b> (Horn '74) $O(n^2)$ Optimal	<b>Tree search</b> (Bratley '71) $O(n n!)$ Optimal
precedence constraints	<b>LDF</b> (Lawler '73) $O(n^2)$ Optimal	<b>EDF*</b> (Chetto et al. '90) $O(n^2)$ Optimal	<b>Spring</b> (Stankovic & Ramamritham '87) $O(n^2)$ Heuristic

Figure 3.17 Scheduling algorithms for aperiodic tasks.

### Exercises

- 3.1 Check whether the Earliest Due Date (EDD) algorithm produces a feasible schedule for the following task set (all tasks are synchronous and start at time  $t = 0$ ):

	$J_1$	$J_2$	$J_3$	$J_4$
$C_i$	4	5	2	3
$D_i$	9	16	5	10

- 3.2 Write an algorithm for finding the maximum lateness of a task set scheduled by the EDD algorithm.
- 3.3 Draw the full scheduling tree for the following set of non-preemptive tasks and mark the branches that are pruned by the Bratley's algorithm.

	$J_1$	$J_2$	$J_3$	$J_4$
$a_i$	0	4	2	6
$C_i$	6	2	4	2
$D_i$	18	8	9	10

- 3.4 On the scheduling tree developed in the previous exercise find the path produced by the Spring algorithm using the following heuristic function:  $H = a + C + D$ . Then find a heuristic function that produces a feasible schedule.

- 3.5 Given seven tasks,  $A$ ,  $B$ ,  $C$ ,  $D$ ,  $E$ ,  $F$ , and  $G$ , construct the precedence graph from the following precedence relations:

$$\begin{array}{ll} A \rightarrow C & \\ B \rightarrow C & B \rightarrow D \\ C \rightarrow E & C \rightarrow F \\ D \rightarrow F & D \rightarrow G \end{array}$$

Then, assuming that all tasks arrive at time  $t = 0$ , have deadline  $D = 25$ , and computation times 2, 3, 3, 5, 1, 2, 5, respectively, modify their arrival times and deadlines to schedule them by EDF.