ALGORITHMS FOR MINIMAL-LENGTH SCHEDULES

by M. DAVIO and A. THAYSE

Abstract
One develops a non-exhaustive algorithm giving optimal schedules of a combinational algorithm and for a fixed number \( m \) of processors. Several arguments such as e.g. partial symmetries in the precedence graph, saturation of the processors, timing constraints on the tasks etc. are used in order to reduce as drastically as possible the computation needed in the selection of an optimal schedule.

1. Introduction

Objectives and motivations of scheduling theory may be formulated in very general terms as follows (we quote Coffman (ref. 2)). The scheduling problems assume a set of resources or processors and a set of tasks which is to be serviced by these resources. Based on prespecified properties of and constraints on the tasks and resources, the problem is to find an efficient algorithm for sequencing the tasks to optimize some desired performance measure. The only measure that will be considered in this paper is the schedule length, i.e. the maximum time spent in the system by the tasks (another measure which is currently considered is the mean time spent in the system by the tasks 2)). The models of the problem we analyse are deterministic in the sense that all information governing the scheduling decision is assumed to be known in advance. In particular, the tasks and all information describing them are assumed to be available at the outset, which we normally take as time \( t = 0 \).

The scheduling model, from which subsequent problems are drawn, will now be described in a more systematic way.

The system of tasks has the structure of a combinational algorithm \((T_0, \prec)\) which can be defined as follows.
1. \( T_0 = \{t_1, t_2, \ldots, t_n\} \) is a set of tasks to be executed.
2. \( \prec \) is an irreflexive partial order defined on \( T_0 \) which specifies operational precedence constraints. In particular \( t_i \prec t_j \) (read as \( t_i \) precedes \( t_j \)) means that task \( t_i \) has to be fully performed before \( t_j \) can start.

The partial order \( \prec \) is conveniently represented as a directed acyclic graph (DAG) with non redundant (or transitive) arcs, called the precedence graph of the system.

The precedence graph is built as the Hasse diagram of the precedence relation.
Algorithms for minimal-length schedules

More precisely, we say that \( t_i \) covers \( t_j \) iff

\[ t_i \prec t_j \text{ and } \exists t_k : t_i \prec t_k \prec t_j. \]

The precedence graph is then built on \( T_0 \) as set of nodes. One draws an arrow from \( t_i \) to \( t_j \) iff \( t_i \) covers \( t_j \). It is a DAG.

We shall borrow some terminology from graph theory and use freely the words *predecessor*, *immediate predecessor*, *successor*, *immediate successor*, *initial node* (node without predecessors) and *terminal node* (node without successors).

We shall refer in the sequel to another graph easily constructed from the precedence graph: the *closed precedence graph* is obtained from the precedence graph by introducing two additional nodes, called BEGIN and END respectively and by drawing an arrow from the BEGIN node to the initial nodes and to the END node from the terminal nodes. The *closed precedence graph* \((T, \prec)\) of a combinational algorithm \((T_0, \prec)\) is thus formed by the nodes

\[ T = T_0 \cup \{B, E\}, \quad (B = \text{BEGIN and } E = \text{END}), \]

and by the arcs corresponding to the precedence relation described hereabove.

For any task \( t \in T \), one defines the *predecessor set* \( P(t) = \{t' \mid t' \in T_0 \text{ and } t \prec t'\} \); in particular, \( P(E) = T_0 \). We denote by \( p(t) \) the cardinality \( \# P(t) \) of \( P(t) \); in particular, \( p(E) = n \). We also denote by \( P_0(t) \) the *set of immediate predecessors* of task \( t \) and by \( p_0(t) \) the cardinality of that set. Similarly, for any task \( t \in T \), one defines the *successor set* \( S(t) = \{t' \mid t' \in T_0 \text{ and } t \prec t'\} \); in particular, \( S(B) = T_0 \). We denote by \( s(t) \) the cardinality \( \# S(t) \) of \( S(t) \); in particular \( s(B) = n \). We also denote by \( S_0(t) \) the set of immediate successors of task \( t \) and by \( s_0(t) \) the cardinality of that set.

So far we avoided to refer in any precise way to the computation time of the tasks \( t_i \). That information was indeed irrelevant to the definitions of precedence and of the associated graphs.

**Assumption 1.** In what follows, we assume that all the tasks \( t_i \) which belong to the task system \( T_0 \) have identical durations. That common duration is chosen as time unit; accordingly, the systems enjoying that property are called *unit execution time* (or UET) systems.

We now associate with each node \( t_i \) of the closed precedence graph two epochs: *early execution time* and *late execution time*. Under the assumption 1 it is easy to adapt to the closed precedence graph the critical path method used in operations research \(^8,9\). We assume that the computation is started with the BEGIN node at \( t = 0 \). The *early execution time* \( e(t_i) \) of the task \( t_i \) is then the first time instant at which task \( t_i \) may be finished, taking into account its
place in the precedence graph and assuming that the number of available processors is arbitrarily large. In particular, the early execution time of the END node: \( e(E) \) is the minimum computation time of the combinational algorithm augmented by one. Let us now impose ourselves to complete that computation of \( E \) at target time \( F \); one will then associate with the task \( t_i \) the \emph{late execution time} \( l_f(t_i) \) which is the last time instant at which task \( t_i \) may be finished to insure the completion of the computation of the tasks of \( T_0 \) at the time \( F - 1 \). Some tasks \( t_i \) in the graph are such that \( e(t_i) = l_f(t_i) \); these tasks are called critical tasks for the target time \( F \).

We now briefly describe an algorithm computing the early execution and late execution times of a closed precedence graph.

\textit{Algorithm 1}

A. \textit{Computation of the early execution times}

\textit{Step 0.} The node \( B \) is marked with 0.

\textit{Step k.} All the hitherto unlabelled nodes all of whose predecessors have been marked in previous steps receive the mark \( k \).

\textit{Step t.} The node \( E \) is marked \( L \).

B. \textit{Computation of the late execution times}

\textit{Step 0.} The node \( E \) is marked \( F \) (\( F \geq L \)).

\textit{Step k.} All the hitherto unlabelled nodes all of whose successors have been marked in previous steps receive the mark \( F - k \).

\textit{Step L + 1.} The node \( E \) is marked \( F - L \).

The above algorithm 1 allows us thus to associate with each task \( t_i \) a time interval

\[ I_f(t_i) = [e(t_i), l_f(t_i)]. \]

An \emph{acceptable labelling} of the precedence graph is then a mapping associating with each task \( t_i \) an epoch \( \psi(t_i) \) such that

\[ \psi(t_i) \in I_f(t_i) \text{ and } t_i < t_j \Rightarrow \psi(t_i) < \psi(t_j). \]

To any acceptable labelling \( \psi \), we shall associate two parameters

(a) its \emph{length} \( F \) (the target time),

(b) its \emph{width} \( W(\psi) \) defined by

\[ W(\psi) = \max \left\{ \# \{ t_i \} \mid \psi(t_i) = k \right\}. \]

It is now easy to formulate the main problem handled by the theory of scheduling in the frame delineated by the above definitions and concepts. Scheduling theory studies the execution of a combinational algorithm \( (T_0, \preceq) \) on a finite
number $m$ of computation resources (the processors). Accordingly, its goal may be formulated as follows. Obtain a minimum length acceptable labelling such that

$$W(\psi) \leq m.$$ 

The retained acceptable labelling will be called the schedule. Informally, a schedule or assignment for a given combinational algorithm is thus the description of the work done by each processor as a function of time. Of course, the schedule must not violate the precedence relations given by the partial order $\prec$.

In what follows, we shall discuss more deeply various methods of obtaining minimum length schedules under the following three additional hypotheses.

Assumption 2. No upper bound is imposed on the number of memory places required to store intermediate computation results.

Assumption 3. A task, once initiated cannot be interrupted, i.e. it must be allowed to run to completion (non-preemptive scheduling).

Assumption 4. All the processors work at the same speed, and this speed is discretized with the restriction that each processor can execute one task during one time unit.

The above presentation and assumptions immediately suggest various enumerative techniques for obtaining optimal schedules. One could e.g. impose extrinsic precedence constraints on the algorithm to reduce the corresponding width. Such a technique was proposed by Fernandez and Lang

Algorithm 2
Step 1. For each task $t_i$, obtain the execution interval

$$I_e(t_i) = [e(t_i), l_e(t_i)]$$

by the algorithm 1.

Step 2. Set $F = e(F)$.

Step 3. Enumerate the acceptable labellings corresponding to the present value of $F$. If one of these labellings has width smaller than the number $m$ of processors, exit. Otherwise, go to step 4.

Step 4. Set $F := F + 1$; go to step 3.

That enumerative approach at once suggests that any general scheduling algorithm has a high computational complexity. An algebraic proof of this
M. Davio and A. Thayse

fact has been given by Ulman 6) who showed that most optimization problems raised by scheduling theory belong to a class of combinatorial problems that have long defied attempts to find a non-enumerative solution. This class of problems has been referred to as \textit{NP-complete} or \textit{polynomial complete} and the interesting property of this class implies the existence of a polynomial solution for any member of the class. Given the size of the class, one may be quite pessimistic about the existence of such a polynomial solution.

Taking into account the above restrictions, there are basically two cases for which polynomials algorithms are known:

(a) when the graph $(T_0, \prec)$ of the combinational algorithm is a tree;
(b) when there are only two processors available.

The case (a), i.e. in which the precedence graph is a rooted directed tree and the number of processors is arbitrary has been considered by Hu 7) who gives an almost linear time algorithm for determining a minimal length schedule for such a task system. The case (b), i.e. when there are only two processors available has been considered by Coffman and Graham 1) who gave an $O(n^2)$ algorithm for this case.

From a practical point of view, scheduling theory may thus assign to itself the following two complementary goals.

(a) Obtain as efficient as possible methods in order to reduce the enumeration required to obtain an optimal schedule. That approach will be followed in sec. 2 of this paper.
(b) Obtain \textit{efficient} scheduling algorithms (i.e. algorithms whose number of steps is a polynomial in the number of tasks to be scheduled).

The preceding discussion, however, shows that there is little hope of discovering such algorithms in the general cases, and this explains why the polynomial algorithms known today are only optimal for restricted classes of problems. This is the case for list scheduling (see Coffman 2) and for the Schindler–Ludtke algorithm 5).

In the present paper we investigate a new approach to the problem of building optimal schedules for a system of $m$ ($m$ is arbitrary) processors. The main idea on which the algorithm developed in the present paper is grounded is the following one.

One develops an algorithm giving in a systematic way the set of all the schedules (optimal or not) of a combinational algorithm and for a fixed number $m$ of processors. Several arguments such as e.g. partial symmetries in the precedence graph, saturation of the processors, timing constraints on the tasks etc. are then used in order to reduce as drastically as possible the computation needed in the selection of an optimal schedule.
2. Restricted enumeration scheduling

2.1. Basic enumeration principle

Consider the closed precedence graph \((T, \prec)\) of a combinational algorithm. The basic enumeration procedure consists in forming step by step the list of acceptable labellings whose width remains smaller than the number \(m\) of processors.

That procedure is described by the following algorithm 3 whose body (steps 2 to 6) accepts as data a family

\[
\mathcal{T}(t) = \{T_t\}; \quad (T_t \subseteq T),
\]

of task sets, each of which may be realized on the \(m\) processors within \(t\) units of time and produces as result a new family \(\mathcal{T}(t + 1)\) of task sets, each of which may be realized on the \(m\) processors within \((t + 1)\) units of time.

**Algorithm 3** (enumerative scheduling)

**Step 1.** Set \(t := 0\); set \(\mathcal{T}(0) = \{\{B\}\}\).

**Step 2.** Let \(\mathcal{T}(t) = \{T_t\}\) (FORK). For each \(T_t \in \mathcal{T}(t)\):

**Step 3.** Compute the set \(H(T_t)\) of all the nodes all of whose predecessors belong to \(T_t\).

**Step 4.** If \# \(H(T_t) < m\) : \(T_t := T_t \cup H(T_t)\). Go to step 5. If \# \(H(T_t) = \mu > m\), form the \(\binom{\mu}{m}\) subsets \(H_j(T_t)\) of \(H(T_t)\) having cardinality \(m\) and replace \(T_t\) by a family of task sets according to

\[
T_t := \{T_t \cup H_j(T_t) \mid j = 1, 2, \ldots, \binom{\mu}{m}\}.
\]

go to step 5.

**Step 5.** (JOIN) \(\mathcal{T}(t + 1) := \cup T_t\), where \(T_t\) are the task sets or families of task sets obtained during step 4. Form \(\mathcal{T}(t + 1)\) by deleting in \(\mathcal{T}(t + 1)\) task sets that are included in other task sets.

**Step 6.** If \(T_0 \in \mathcal{T}(t + 1)\), then exit, else go to step 2, with \(t := t + 1\).

**Comments.** (1) Once \(T_0\) has been obtained as, say \(T_0 \in \mathcal{T}(p)\), the corresponding schedule is readily available. Indeed, each element of \(\mathcal{T}(p)\) is obtained by the concatenation of \(H_i(T_0)\)'s obtained during the successive iterations of the algorithm. One easily recognizes that after the \(k\)th iteration, any element of \(\mathcal{T}(p)\) may be written as

\[
\{B, (H_i)_1, (H_i)_2, \ldots, (H_i)_k\},
\]
where \((H_j)_j\), \(1 \leq j \leq k\), is a \(m'\)-tuple \((m' \leq m)\) of tasks obtained at the \(j\)th iteration \((t = j)\) of the algorithm. If one now assigns the \(m'\) tasks of \((H_j)_j\) to be performed by the \(m\) processors during the time interval \((j, j + 1)\), and if \(T_0 \in \mathcal{T}(p)\), one then easily recognizes that the above set of \(m'\)-tuples represents a schedule of length \(p\).

(2) Clearly, if \(n\) is the least integer such that \(T_0 \in \mathcal{T}(p)\), the corresponding schedule is optimal. Algorithm 3, however, does not lead to all the optimal schedules since the deletion operation in step 3 favors an early execution of the partial tasks in \(T_0\).

Example 1. Consider the precedence graph given in fig. 1a. Assume \(m = 4\); the obtention of the consecutive families of task sets \(\mathcal{T}(t)\) by means of algorithm 3 is illustrated graphically in fig. 2. One obtains the single optimal schedule given in fig. 1b. Observe that in such a graph, a particular set \(T_i(t)\) may have more than one predecessor. Observe also that a particular set \(T_i(t)\) may have no successor: this happens when the deletion clause in step 5 is active. Our example also illustrates the fact that algorithm 3 does not yield all the optimal solutions; for example the optimal schedule in figure 1c is not obtained by the algorithm. The latter thus already contains enumeration reducing features.

![Fig. 1. (a) Precedence graph. (b,c) Optimal schedules.](image-url)
2.2. Task equivalence

In the remaining part of this section, we introduce various methods allowing one to reduce the enumeration involved in algorithm 3.

Two partial tasks $t_i$ and $t_j$ of a combinational algorithm are equivalent iff they have identical predecessor and successor sets. Similarly, we say that two schedules are equivalent iff they may be deduced from each other by permutation of equivalent partial tasks. As usual, the simplification resulting from an equivalence detection consists in the fact that one only needs to consider a single representative schedule per equivalence class. An easy way to insure a proper bookkeeping consists in representing by the same symbol all the partial tasks in a given equivalence class and to account for the number of occurrences of that symbol instead of the individual tasks themselves.

Let us for instance replace in figure 1a, the equivalent partial tasks 3, 4 and 5 by the symbol $A$ and the equivalent partial tasks 8, 9, 10 and 11 by the symbol $C$. Figure 3a illustrates the research of an optimal schedule for $m = 3$. Figures 3b and c represent the optimal schedule obtained.

Remarks. (1) We shall not discuss here the task equivalence detection itself. An algorithm for obtaining that result would easily be set up if the precedence graph were e.g. described by its adjacency matrix.

(2) The comparison of figs 1 and 3 illustrates the fact that a reduction of the number of processors does not necessarily increase the required computation time.
2.3. Critical path approach

2.3.1. General principle

The main purpose of this section will be to develop criteria that reduce as much as possible the size of algorithm 3. The scheme that will be proposed in order to reach that goal will be the following one. The algorithm 3 remains the central part in the determination of optimal schedules, however the number of task subsets $H_j(T_i)$ to be examined during the successive iterations will be reduced as much as possible according to criteria to be defined further on.

Let us first recall that in sec. 1 we associated with each task $t_i$ an early execution time $e(t_i)$, a late execution time $l_f(t_i)$ and an execution interval $I_f(t_i) = [e(t_i), l_f(t_i)]$. These concepts were introduced independently of the number $m$ of processors or otherwise stated, it was assumed that the number $m$ of processors was arbitrarily large. We now enlarge the above definitions to the case of a finite number $m$ of processors.

The early execution time $e_m(t_i)$ of the task $t_i$ is the first time instant at which task $t_i$ may be finished, taking into account its place in the precedence graph and assuming that the number of processors is $m$. In particular, the early execu-
tion time of the END node $e_m(E)$ is the length of the optimal schedules augmented by one. One will also associate with the task $t_i$ a late execution time $l_m(t_i)$ which is the last time instant at which task $t_i$ must be finished to insure the completion of the tasks of $T_0$ in a minimum length schedule (of length $e_m(E) - 1$). The optimal execution interval for a task to be inserted in an optimal schedule is

$$I_m(t_i) = [e_m(t_i), l_m(t_i)].$$

It is clear that the knowledge of $I_m(t_i) \forall i$ would be a decisive step in the building of optimal schedules. Further on, one will construct algorithms for obtaining as good as possible approximations for both $e_m(t_i), l_m(t_i)$ and thus for $I_m(t_i)$; these approximations will be written

$$I_m^*(t_i) = [e_m^*(t_i), l_m^*(t_i)].$$

The following algorithm 4 constitutes an improvement of algorithm 3; it may be considered as an interconnection of algorithms 1, 2 and 3.

**Algorithm 4**

**Step 1. (time interval approximation algorithm)**

For each task $t_0$ of $T_0$ obtain an approximation $I_m^*(t_i)$ of the optimal execution interval $I_m$. The simplest type of *time-interval approximation algorithm* is algorithm 1 which produces the approximation $I_F(t_i)$, i.e.

$$I_m^*(t_i) = I_F(t_i) \quad \forall t_i \in T.$$

(More elaborate types of time-interval approximation algorithms producing better approximations for $I_m(t_i)$ will be developed in sec. 2.3.2.). If algorithm 1 is used one has thus

$$e_m^*(t_i) = e(t_i); \quad l_m^*(t_i) = l_F(t_i).$$

**Step 2.** $F = e_m^*(E). ( = e(E)$ if the time-interval approximation algorithm is algorithm 1).

**Step 3. (compatibility-checking algorithm)**

Check if the time interval approximation is compatible with the number $m$ of processors.

The simplest type *compatibility-checking algorithm* that will be used for testing the compatibility of the approximation with the number of processors is the following.

An approximation determines a set of critical nodes, i.e. nodes for which the early execution time is equal to the late execution time.

If there exists at least one time $t$ for which the number of critical tasks is larger
than \( m \), the approximation can evidently never lead to the building of a schedule with \( m \) processors; it will be said to be incompatible with \( m \). More elaborate types of compatibility checking algorithms will also be considered further on in sec. 2.3.2.

If the approximation is compatible, go to step 5; otherwise go to step 4.

**Step 4.** \( F := F + 1 \); go to step 3.

**Step 5.** Perform algorithm 3 with the following restriction. At each iteration (an iteration of algorithm 3 is characterized by its index \( t \)) retain only the task sets \( H_j(T_i) \) whose elements \( t_j \) have \( t \) in their execution interval, i.e. \( t \in I^*_m(t_k) \forall t_k \in H_j(T_i) \). If for a given time index \( t \), \( \exists H_j(T_i) \) satisfying the above condition go to step 6; otherwise continue algorithm 3.

**Step 6.** Set \( F := F + 1 \); go to step 5.

The main idea underlying the statement of algorithm 4 is the following one. During each iteration of algorithm 3, step 4 produces task sets \( H_j(T_i) \); to each of these task sets corresponds a new branch of the algorithm to be explored during the next iteration (this can e.g. easily be verified by considering the precedence graph of the schedule system). Only a subset of these branches leads to optimal schedules. Hence the need for detecting as soon as possible the sets \( H_j(T_i) \) which never will generate optimal schedules; these sets will then be developed in the future computations. The ways used in algorithm 4 for differentiating acceptable from non acceptable sets \( H_j(T_i) \) are

1. the approximation \( I^*_m(t_k) \) of the optimal time interval,
2. the compatibility test.

It turns out now that algorithm 4 will converge more rapidly (i.e. will contain a less number of iterations) if

1. \( I^*_m(t_k) \) approximates in a better way the optimal execution interval \( I_m(t_k) \);
2. the test for compatibility is not only grounded on the number of critical tasks.

**2.3.2. Elementary criteria for improving algorithm 4.**

We shall first present a method for labelling the nodes of the closed precedence graph by a couple of integers:

\[ [e^*_m(t_i), l^*_m(t_i)]. \]

As in the method of algorithm 1, both integers will be obtained by recursive forward and backward processes, respectively. Furthermore, \( e^*_m(t_i) \) will represent the approximation of a lower bound on the execution time of task \( t_i \) in an optimal schedule while \( l^*_m(t_i) \) will represent an approximation of an upper bound of that time. As distinct from the method of algorithm 1, these two bounds will be made dependent on the number \( m \) of available processors.
We first discuss two elementary criteria which may be useful to sharpen the lower bound \( e_m(t_i) \). Similar considerations would obviously apply to the upper bound.

**Criterion 1.** If the node \( t_i \) has the immediate predecessor \( t_j \), then

\[
e_m^*(t_i) \geq e_m^*(t_j) + 1.
\]

**Criterion 2.** If the node \( t_i \) has \( P_0(t_i) \) predecessors in \( T_0 \), then

\[
e_m^*(t_i) \geq \left\lceil \frac{p_0(t_i)}{m} \right\rceil + 1.
\]

(The symbol \( \lceil a \rceil \) denotes the smallest integer not smaller than \( a \).)

Accordingly, our labelling algorithm 1 would become the following one.

**Time interval approximation algorithm**

**Algorithm 5.** (Forward process: computation of the early execution times)

**Step 1.** Label \( B \) with \( e_m^*(B) = 0 \).

**Step 2.** Label all the hitherto unlabelled tasks all of whose predecessors have been labelled according to the rule

\[
e_m^*(t_i) = \max_{t_j \in P_0(t_i)} \left\{ e_m^*(t_j) + 1, \left\lceil \frac{p_0(t_i)}{m} \right\rceil + 1 \right\}
\]

**Step 3.** If \( E \) is labelled exit; else go to step 2.

Let us now choose a target time \( F \geq e_m^*(E) \).

**Algorithm 5.** (Backward process: computation of the late execution times)

**Step 1.** Label \( E \) with \( l_m^*(E) = F \).

**Step 2.** Label all the hitherto unlabelled nodes all of whose successors have been labelled according to the rule

\[
l_m^*(t_i) = \min_{t_j \in S_0(t)} \left\{ l_m^*(t_j) - 1, F - \left\lceil \frac{s_0(t)}{m} \right\rceil - 1 \right\}
\]

**Step 3.** If \( B \) is labelled, exit; else go to step 2.

The labelling obtained by the above algorithm 5 may effectively be used to cut down the remaining enumerative work of algorithm 4. This is illustrated by the following example 2.

Before dealing with example 2, we shall present a model allowing us to represent in a relatively compact form optimal schedules that are deduced from algorithms 3 or 4.

This set of optimal schedules may be viewed as a new combinational al-
algorithm, namely the schedule system. This schedule system is built up as follows.

Let \((H_i)\) be a set of tasks obtained at step 4 of the algorithm 3 and at its \(j\)th iteration; the schedule system is a combinational algorithm \((H, \sqsubseteq)\) which can be defined as follows.

1. \(H = \{(H_i)\}\);
2. \(\sqsubseteq\) is an irreflexive partial order defined on \(H\) which specifies operational precedence constraints, i.e.

\[
(H_i)_j \sqsubseteq (H_k)_{j+1}
\]

iff

\[
(H_k)_{j+1} = H_k(T_t \cup (H(T_i)))
\]

Consider figures 2 and 3; one verifies that the elements of \(H\) are nothing but the labellings of the arrows in these figures. As will be seen in example 2 below, the schedule system allows us to represent graphically a set of optimal schedules in a relatively compact form even for elaborated examples.

Example 2. Consider the task system whose closed precedence graph is given in fig. 4; consider moreover the computation of this system of tasks by means of a set of 3 processors. The associated schedule system is described by its precedence graph in fig. 5. Let us also recall that the number of nodes in this graph may be considered as a measure of the number of computations performed by algorithm 3 during the determination of optimal schedules.

Let us now compute approximations for the optimal time intervals of the nodes of fig. 4 by means of algorithm 5. The results of algorithm 5 have been written down in fig. 4 (for \(F = e_m^\ast(E) = 10\): each node \(t_j\) is followed by its time interval written \([e_m^\ast(t_j), l_m^\ast(t_j)]\). One sees in fig. 4 that 24 and 27 are critical tasks at \(t = 1\), 22 is a critical task at \(t = 3\), 18, 19 and 20 are critical tasks at \(t = 4\), 13 and 14 are critical tasks at \(t = 5\) and 10 is a critical task at \(t = 6\). This system of critical tasks is coherent with the number \(m\) of processors since there does not exist a time \(t\) for which the number of critical tasks is larger than 3.

Let us now describe how the information given by algorithm 5 may be used to reduce the computations in algorithm 3. Since 24 and 27 are critical tasks at \(t = 1\), the sets \(H_i(B)\) obtained during the first iteration of algorithm 3 must necessarily contain both tasks 24 and 27; one has thus (24 and 27 are equivalent tasks labelled \(G\))

\[
H_i(B) = \{\{23, G, G\}, \{A, G, G\}\}
\]

(these tasks have been written in a black format in fig. 5) and, as it can be
Algorithms for minimal-length schedules

Fig. 4. Closed precedence graph of example 2 for 3 processors.

Fig. 5. Schedule system for the example 2.
verified in fig. 5, the number of sets of tasks to be examined during the second iteration of the algorithm is reduced from 6 to 2. Similarly, since 18, 19, and 20 are critical tasks at $t = 3$, there is only one acceptable set $H_i(F, F)$ (see fig. 5) while otherwise, four sets $H_i(F, F, 22)$ should be examined at the 4th iteration of algorithm 3. Consider fig. 5: the number of task sets $H_i(T_p)$ generated by algorithm 3 is the number of nodes in this figure, that is 42. The computation of algorithm 5 reduces the tasks sets $H_i(T_p)$ to those which have been written in a black format in fig. 5. Algorithm 4 (which is an improvement of algorithm 3 by means of algorithm 5) allows us to compute only 15 task sets instead of 42.

The first step of algorithm 4 is formed by a time interval approximation algorithm which gives an approximation of the optimal intervals for the tasks $t_i \in T_0$. It turns out that for a better approximation the algorithm 5 will converge more rapidly; the approximation given by algorithm 4 constitutes an improvement with respect to that of algorithm 1. Further developments concerning these approximations will be discussed in sec. 4.

Time interval approximation algorithms are formed by a forward process and a backward process. Before starting at the step 5 (of algorithm 4), the computation of task sets $H_i(T_p)$, it is preferable to detect the interval approximations that are incompatible with the number $m$ of processors. This is done at the step 3 of algorithm 5 by a compatibility checking algorithm. As for the time interval approximation algorithms, there exist several types of compatibility checking algorithms giving less or more complex kinds of checkings; this will be developed herebelow.

As quoted in the beginning of this section, the simplest type of compatibility algorithm is grounded on the counting of critical nodes at each time instant; it may briefly be summarized as follows.

(Compatibility-checking algorithm)

Step 1. Let $F$ be the chosen target time.

Step 2. Check if there exists some time instant $t$ for which the number of critical tasks exceeds the number of processors. If such a time instant exists set $F := F + 1$ and go to step 1; otherwise go to step 3.

Step 3. := next step in the computation process.

The above algorithm may be improved in the following way.

Algorithm 6. (Compatibility-checking algorithm)

Step 1 and Step 2 are identical to the compatibility-checking algorithm quoted hereabove.
Step 3. Let $m' \leq m$ be the number of tasks that are critical at the time $t$; the number of available processors (for the other tasks) at $b$ is thus $(m - m')$ instead of $m$. Verify whether this reduction in the number of available processors produces new critical tasks. If new critical tasks are produced go to step 2; otherwise go to step 4.

Step 4 := next step in the computation process.

The algorithm 6 may effectively be used to cut down the remaining enumerative work of algorithm 4. This is illustrated by continuing the example 4.

Example 2 (continued). Consider the task system whose closed precedence graph has been given in fig. 4 but assume now that the computation of this system is performed by means of a set of 4 processors. Let us compute approximations for the optimal time intervals of the tasks by means of algorithm 5. The results of algorithm 5 have been written down in fig. 6. In view of the results of fig. 6, let us start with a compatibility-checking algorithm:

<table>
<thead>
<tr>
<th>time</th>
<th>critical nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24,25,26,27;</td>
</tr>
<tr>
<td>2</td>
<td>22;</td>
</tr>
<tr>
<td>3</td>
<td>18,19,20;</td>
</tr>
<tr>
<td>4</td>
<td>13,14;</td>
</tr>
<tr>
<td>5</td>
<td>10.</td>
</tr>
</tbody>
</table>

Since the maximum number of critical nodes for a given time $t$ is not larger than 4 (number of processors) the simplest type of compatibility-checking algorithm is satisfied. Assume that this type of algorithm is used at step 3 of algorithm 4; the results of the computation are gathered in fig. 7. In view of fig. 7, one sees that the step 3 is stopped at the fifth iteration which detects incompatibilities in each of the sets $H_i(T_p)$. Now these incompatibilities should also be detected at once (without entering in step 3 of algorithm 4) by using the algorithm 6 as compatibility checking algorithm. Indeed, the following information may be deduced from fig. 6. Since there are 4 critical nodes at $t = 1$, the node 23 is necessarily critical at $t = 2$; hence: 21 is critical at $t = 3$, and 15, 16 and 17 are critical at $t = 4$, which is impossible, since there were already two nodes, namely 13 and 14 which were critical at this time. The system with $F = 8$ is thus incompatible and new computations must be started with $F = 9$. Let us finally also note that the graph of fig. 7 constitutes a subgraph of the schedule system which can be deduced from algorithm 3.
Fig. 6. Closed precedence graph of example 2 for 4 processors.

Fig. 7. Subgraph of the schedule system.
The effectiveness and the weakness of algorithm 4 clearly appear in the above example. With respect to algorithm 3, the algorithm 5 introduces two criteria, namely the compatibility criterion and the critical path criterion. These two criteria allow us to consider only a subgraph of the schedule system, but on the other hand the verification of these criteria needs extra computations. It is thus difficult to have a definite opinion relative to the simplification of the computations introduced by the two criteria and for any type of combinational algorithm. It may, however, be mentioned that there exist several families of graphs for which the simplification criteria drastically reduce the amount of computation. Consider e.g. the family of graphs represented in fig. 8. One verifies that for

\[ q = \beta m (m - 2) - 1; \quad p = \beta m \quad (\beta \text{ is an arbitrary integer}) \]

the critical path criterion renders all the nodes critical while the enumerative method of algorithm 3 produces an excessive amount of computation.

The compatibility criterion and the critical path criterion algorithms considered in this section are representatives of two general types of algorithms. More elaborated representatives of these algorithms will be considered in sec. 2.3.3. below.

2.3.3. Strong criteria for improving algorithm 4

A time interval approximation algorithm is basically an algorithm which, by means of recursive forward and backward processes, associates two integers (defining a time interval) to each node of a task system.

Fig. 8. A class of combinational algorithms.
A first time interval approximation algorithm has been described in sec. 1 (algorithm 1); a second algorithm making use of two criteria (algorithm 5) has been described in sec. 2.3.2.

Let us observe that the two criteria of algorithm 5 actually cooperate to yield as high a value of $e_m^*(E)$ as possible. Basically, criterion 1 is efficient along chains while criterion 2 is efficient against large number of independent tasks appearing at the same level of the graph. The quantity $e_m^*(E)$ obtained as result of the forward process of algorithm 5 is not a tight lower bound of the minimum achievable target time $F_{\text{min}}$. In fact, we may exhibit graphs for which

$$F_{\text{min}} = e_m^*(E) + K$$

with an arbitrarily large $K$. This will be illustrated by the following example.

**Example 3.** The closed precedence graph is displayed in fig. 9. The task system is built in such a way that, in the largest part of the graph, the width is smaller than the number of available processors, while the overall $e_m^*(E)$ is determined by the total number of tasks. The algorithm 5 will yield here, under the assumptions

(a) $m > 2$,
(b) $2p + q > m(p + 2)$

the result

$$e_m^*(E) = \left\lceil \frac{2p + q}{m} \right\rceil + 1$$

while clearly one has

$$F_{\text{min}} = p + \left\lceil \frac{q}{m} \right\rceil + 1.$$
Consider the particular situation \( q = (p + 2)m \). It is easily checked that this assumption satisfies the requirement \((b)\). The latter is indeed equivalent to

\[
q > p(m - 2) + 2m
\]

and

\[
(p + 2)m = p(m - 2) + 2m + 2p > p(m - 2) + 2m.
\]

In that situation

\[
e_m^*(E) = p + 2 + \left\lceil \frac{2p}{m} \right\rceil + 1, \quad \text{and} \quad F_{\text{min}} = p + (p + 2) + 1.
\]

The difference

\[
F_{\text{min}} - e_m^*(E) = p - \left\lceil \frac{2p}{m} \right\rceil
\]

may be made as large as suitable by an appropriate choice of \( p \). Indeed, the inequality

\[
p - \left\lceil \frac{2p}{m} \right\rceil > K
\]

is satisfied by any value of \( p \) fulfilling the condition

\[
p > \frac{2p}{m} + 1 + K
\]

i.e. by

\[
p > \frac{m(K + 1)}{m - 2}.
\]

Observe that the latter condition is only meaningful if assumption \((a)\): \( m > 2 \) holds true. As a matter of fact, that assumption is equivalent to the impossibility of using all of the \( m \) processors during the \( p \) first computation steps.

The lack of tightness of the lower bound \( e_m^*(E) \) is due to the lack of severity of the criteria 1 and 2. In what follows we shall replace these criteria by two stronger criterions respectively.

**Criterion 1**. The set of immediate predecessors of node \( t \) to be labelled may be partitioned according to the values of the labels of these nodes. Let

\[
P_{01}(t) = \{ t' \mid t' \in P_0(t) \quad \text{and} \quad e_m^*(i_i) = l_i \}
\]

and

\[
P_{00}(t) = \# P_{01}(t).
\]
Clearly we should have
\[ e_m^*(t) \geq \left\lfloor l_t + \frac{P_{0}(t)}{m} \right\rfloor. \]

*Criterion 2*. Let now \( t' \in P_0(t) \cup \{B\} \). Consider the interval \([t', t] \) i.e. the set of nodes
\[ \{ n \mid t' < n < t \quad \text{and} \quad t' \neq n \neq t \} \]
Let \( p[t', t] \) represent the cardinality of this set. We have
\[ e_m^*(t) > e_m^*(t') + \left\lfloor \frac{p(t', t)}{m} \right\rfloor + 1. \]

Now, criterion 1* is obviously stronger than criterion 1 since
\[ l_t + \left\lfloor \frac{P_0(t)}{m} \right\rfloor \geq e_m^*(t') + 1. \]

Similarly, criterion 2* is stronger than criterion 2 since it reduces to the latter whenever \( t' = B \).

Algorithms 1, 5 and the criteria 1, 2, 1* and 2* show the existence of families of time interval approximation algorithms giving better and better bounds for the time intervals by means of more and more tedious computations.

One could similarly make obvious the existence of families of compatibility checking algorithms which develop more and more complex compatibility tests by means of more and more intricated arguments.

Two types of compatibility algorithms were developed in sec. 2.3.1.: the compatibility criterion was grounded on the number of critical tasks at any time instant \( t \) of the scheduling process. It turns now out that similar compatibility criteria could be developed on non-critical tasks: consider e.g. the tasks that have the task interval \([e_m^*, l_m^*] \). If the number of these tasks is larger than \( m (l_m^* - e_m^* + 1) \), the labelling is evidently incompatible with the number \( m \) of processors and the target time must be increased.

Another way to generalize the compatibility criteria of sec. 2.3.1. is the following. If at the time \( t \), there are \( m \), tasks that are critical, there remain only \( m - m_t \) processors for the other tasks at that time \( t \) so that in the rules for computing time intervals for these tasks, \( m \) should be substituted by \( m - m_t \) in a more refined type of computation.

3. Conclusion and open questions

In section 3.1 we present a restricted enumeration algorithm for obtaining
optimal schedule(s) for a given combinational algorithm. This basic algorithm is improved in secs 3.2–3.3 by introducing several simplification criteria. One observes, however, that the introduction of these criteria renders the algorithm more tedious to deal with and thus might also in some cases make the computations more cumbersome. One of the problems which remains of interest for future investigations, is to determine classes of combinational algorithms for which the improved algorithm 4 should always result in simplification of the computations with respect to those of the basic algorithm 3. A partial answer to this question has already been given in sec. 3.3.2, where one defines a class of combinational algorithms for which the computation of a time interval (which occurs in the critical path approach) always leads to a simplification of the resulting computations.

MBLE Research Laboratory
Brussels, December 1978

REFERENCES

4) E. Fernandez and T. Lang, Scheduling as a graph transformation, IBM J. Res. Dev. 20, 551-559, Nov. 1976.