Sequencing Aspects of Multiprogramming*

J. Heller

Institute of Mathematical Sciences, New York University, New York, N. Y.

1. *Introduction and Viewpoint*

The large newer computers available today and the computers of the future will all have features of simultaneous operation. Because of this simultaneity, the question of planning a program and the planning of groups of independent programs using the simultaneous operation capabilities of the computer becomes important. All these questions have been grouped under the heading of multiprogramming. In this paper we will discuss the sequencing or scheduling aspects of multiprogramming. Roughly speaking, the sequencing aspects revolve around questions of parts of a computer being idle because the data to be processed in one computer part is still being processed elsewhere in the computer; and if we have a group of independent programs, how can we stagger the parts of the programs through the computer such that some defined objective of all the programs is optimized.

These considerations have been studied partly in a different context: the so-called machine shop scheduling [cf. [7] for a general review and further references]. There are features of similarity between machine shop (MS) scheduling (S) and multiprogramming (MP) scheduling. There are also features of dissimilarity.

In the simple case of MSS [3], each job has a given order of processing on the machines of the MS. This ordering, one for each job, is called a *technological ordering*. The only cases of MSS studied in mathematical detail have assumed that each job goes on each machine at most one time. If we liken the machine shop to the computer, the program to the job, the computer parts—input channels, processing units and output channels—to the machines of the machine shop and the ordering of the subparts of the program on the computer parts, there appears to be a difference between MPS and MSS as described above. Whereas a job (equivalent to a program in MPS) in MSS scheduling goes on each machine (computer part) at most once, the subprograms of a program (job) can return to the computer part (machine) more than once. The latter difference makes for a more complex problem, which has not been studied in mathematical detail in the machine shop scheduling context, but has been discussed under the name of job-lot MSS [7].

In order to gain an insight into MP and see through the maze of intricacies

* Received November, 1960; revised March, 1961. The work presented in this paper was supported by the AEC Computing and Applied Mathematics Center, Institute of Mathematical Sciences, New York University, under Contract AT(30-1)-1480 with the U. S. Atomic Energy Commission.

necessary to describe idle time on a computer part for one program or a schedule of independent programs, we will proceed with simple examples in simple steps. First we will consider flow diagrams; then schedule diagrams, which describe the technological ordering of the program; and finally Gantt or timing diagrams,[1] which describe the actual times to finish a particular subprogram. We will first consider the case of one program. After the one-program case is considered, we will consider the scheduling of many programs as might arise in a single computer.

Our aim, as stated before, will be to formulate the problems of MPS. The complex combinatorial and statistical questions will not be discussed here. The analysis of these questions is left to future investigations.

2. *One-Program Case*

Let us consider a computer which has many processing units, many input channels and many output channels that can operate simultaneously. In order to describe idle times on each computer part[2] we would have to know which program parts (from now on called subprograms) precede which other subprograms and the time it takes to perform each subprogram. A flow diagram does not describe all the needed features; it describes the logical flow of the program, indicating sequential steps with branching indicating possible simultaneous operation. On the other hand, a Gantt or timing diagram indicates the time flow of the subprograms but partly hides those subprograms which "delay" other subprograms. Between these two types of diagrams is another diagram which brings out the features of the possible "delays" between the subprograms and allows for a description which is very useful for combinatorial studies of scheduling. We consider a simple example to illustrate these features.

In Figure 1 we have a flow diagram for a simple problem performed on a computer having two input channels, two processors and an output channel. We indicate when the inputs, processing and outputs are performed. However, we do not clearly distinguish the simultaneous operations; e.g. in box 2 there is a statement about inputs from channels 1 and 2. Presumably they are to be performed simultaneously. On the other hand, some statements in different boxes may indicate simultaneous operation, e.g. statements in boxes 3 and 4.

In order to study the "delays" more clearly, we resort to the construction of a linear graph which indicates the use of each computer part separately and by directed arrows those computer parts which must be finished with a subprogram

[1] Flow diagrams have been discussed in the present context. Schedule diagrams have been discussed in [2] for simple MSS, and are nothing more than diagrams of partially ordered sets [2]. Gantt diagrams are commonly used in machine shop scheduling (cf. [7] and the references given there).

[2] By *computer part* we mean a processing unit, an input channel and an output channel. We could discuss the intuitive features of multiprogramming in terms of other more detailed parts of a computer system. However, this detail is not needed to bring out the mathematical structure, and hence we stay with the input, process, output terminology.

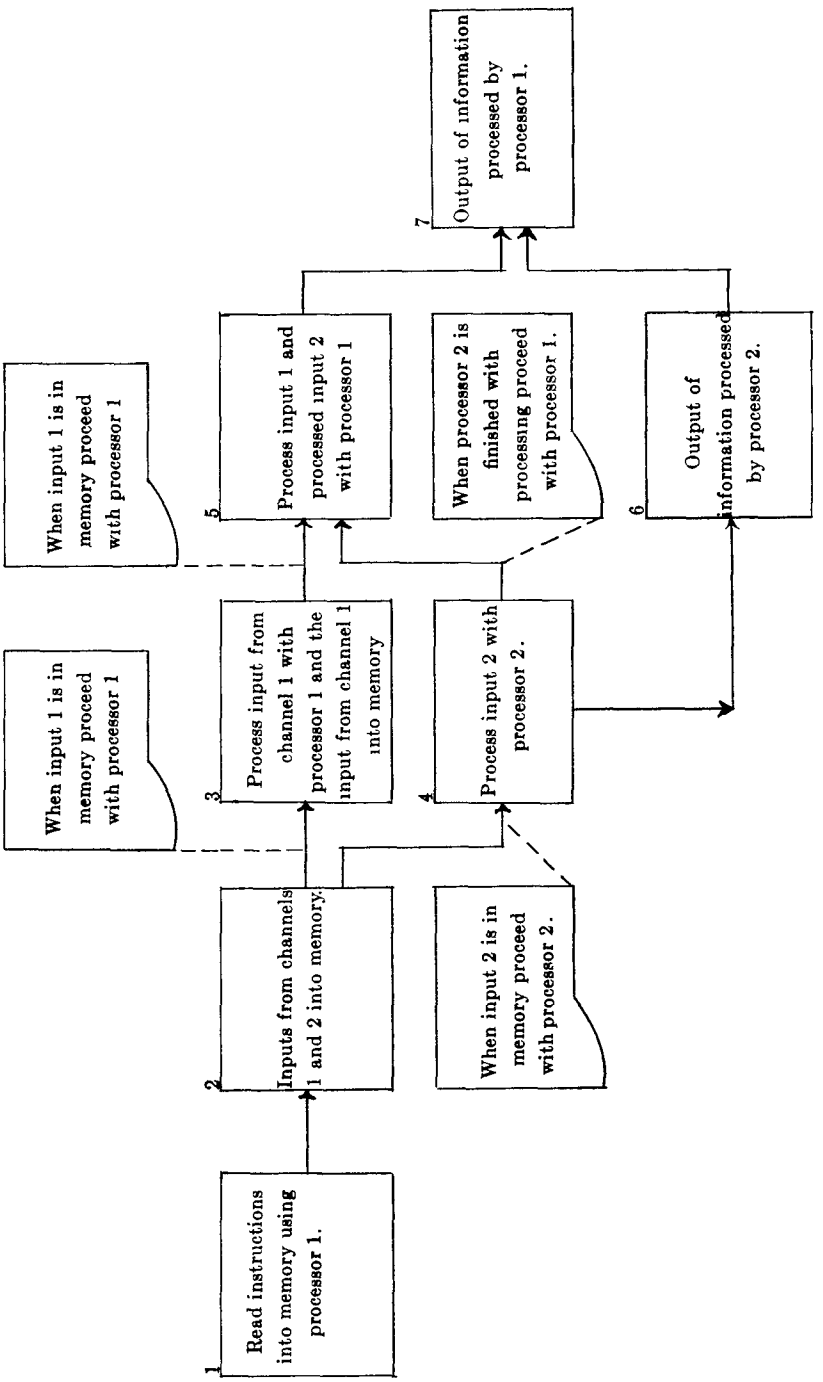

FIG. 1. A flow diagram for a program using simultaneous features of a computer

before another subprogram can start.[3] In Figure 2 we have constructed the linear graph for the flow diagram of Figure 1.

In each node (circle) there is a statement describing a particular subprogram which uses but one computer part. The directed branches leading away from a node (say η) indicates those subprograms which can possibly start when the subprogram for node η is finished. We observe that the directed branches leading into a particular node (say η) come from all the subprograms that can "delay" the subprogram of the node η.

We have indicated on each horizontal line the nodes which use each computer part. On the left is written the computer part. On the right under the heading m (for machine) these computer parts are numbered. Under each node is given a triplet of the form (mji). The m refers to the computer part; j refers to the particular program; i refers to the number of times that the computer part m has been used up to and including the node (mji) of the program j. The triplet (mji) denotes the "jth program on the mth computer part for the ith time." If we were discussing job lot machine shop scheduling, we would say that (mji) means the "jth job on the mth machine for the ith time."

Diagrams of the type given in Figure 2 are *linear graphs*, for the nodes (mji) are a set $\mathcal{J}_j$ and the branches are a set[4] $\mathcal{C}_j$.

The linear graph picture contains most of the information from the discussion of timing. All that is needed is the knowledge of the time t_{mji} to perform each subroutine. The subroutine time corresponds to the processing time in MSS.

It is an easy matter to draw the Gantt or timing diagram from the linear graph. Along each horizontal line, one for each computer part, we draw arrows in order of performance of subroutines proportional to the time of completing each subroutine. The tail of any arrow (say η) is on a vertical line with the head of the arrow which represents the finishing of the last subprogram to "delay" the subprogram represented by the arrow η.

We have drawn the Gantt or timing diagram in Figure 3 for the diagram of Figure 2 using a given set of processing times t_{mji}.

3. *Idle and Subprogram Finishing Times for a Single Program*

A diagram or linear graph embodies the properties of a technological ordering. The idle time before starting a given subprogram and the time to complete a particular subprogram are functions defined on the linear graph of the program. We proceed to express these functions in terms of the processing times t_{mji} of the subprograms.

[3] In [4] diagrams of the type we are about to consider were considered for MSS. However, these diagrams were inferred from the Gantt diagrams. For our purpose the present logical development seems more suitable.

[4] Berge [1] uses X for our set $\mathcal{J}_j$ and U for our set $\mathcal{C}_j$. The $\mathcal{J}_j$ is the terminology taken from MSS and the set $\mathcal{C}_j$; the covering relations have been implied in the MSS study of reference [4]. Unfortunately, to confuse matters further, Sisson [8] uses $\mathfrak{M}_j$ for our $\mathcal{J}_j$.

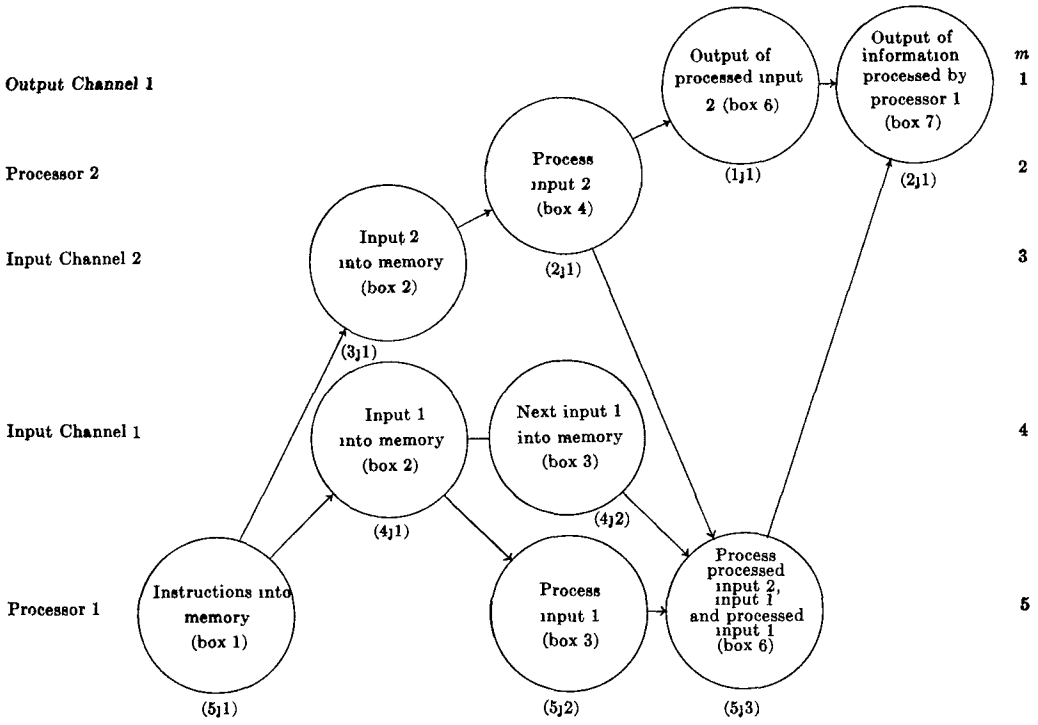

FIG. 2. The diagram for the flow diagram of Figure 1

In order to define these functions we must have a notation to keep track of the order of performing the subprograms. We use the binary relation $\leqslant$, meaning "is started before or at the same time as" relating two subprograms. For example, $(4j2) \leqslant (5j3)$ means the jth program on the 4th computer part for the 2nd time is started before or at the same time as the jth program on the 5th computer part for the 3rd time. From a diagram such as that given in Figure 2, the binary relations can be written down, there being one for each branch and those obtain transitively from these order relations. The resulting set with their ordering relations $(\mathcal{J}_j, \mathcal{C}_j)$ is *partially ordered*.[5]

In previous mathematical studies [cf. [3] and [4] for examples and further references], the technological ordering has always been taken as a simple ordering [5]. In these cases an object j is processed sequentially and if an object is processed by itself there is no idle time from machine to machine. However, in the present case there may be idle time.

[5] The definitions of the various "ordered" sets that we will have occasion to refer to will be those given by Birkhoff [2]. By *partial ordering* we mean that not all elements are related and that those elements that are related satisfy a transitive relation. By a *simple ordering* we mean that every distinct pair of elements is related.

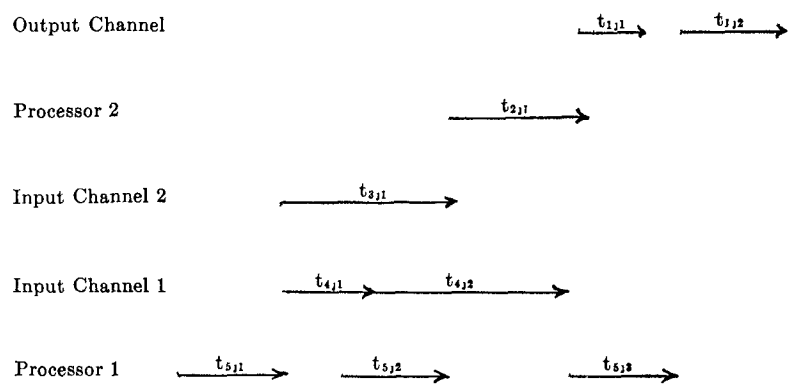

FIG. 3. The Gantt or timing diagram for the program of Figure 1

One other binary relation of use in our discussion is a covering relation.[6] If two modes (mji) and $(m'j'i')$ are connected by a directed branch going from (mji) to $(m'j'i')$, we say "(mji) is covered by $(m'j'i')$" and we write this expression as $(mji) \rightarrow (m'j'i')$.

We are now in a position to give expressions for the idle time and completion time for a given subroutine. Each of these functions is defined in terms of the other and the processing times; and then we give conditions under which we can solve for these functions in terms of the processing times.

Let us denote the idle time before processing the (mji) as I_{mji}; let us denote the completion time of the (mji) as $T(mji)$. In order to obtain an expression for the idle time we must know which subprograms can "delay" a given subprogram. The question of the possible "delay" of a given subprogram, say (mji), depends on the completion times of all those subprograms which are covered by the subprogram (mji). From those subprograms covered by (mji), we define the last one to be completed as $\max_{m'} T(m'j'i' \rightarrow mji)$.

We now define the completion time and idle time in terms of each other and the known processing times t_{mji}. We take the completion times of (mji) as

$$T(mji) = \sum_{\substack{i' \ni \\ (mji') \leq (mji)}} (t_{mji'} + I_{mji'}) \tag{3.1}$$

which assumes that a subprogram is started as soon as possible, subject only to the "delay" that can occur due to the technological ordering. We take

$$\max_{m'} T(m'ji' \rightarrow mji) = \max_{m'} \sum_{\substack{i' \ni \\ (m'ji') \leq (m'ji'') \rightarrow (mji)}} (t_{m'ji'} + I_{m'ji'}) \tag{3.2}$$

[6] B. Giffler (cf. [7]) calls this relation "next precedence." In [3] explicit use was made of the covering relation in the derivation of the idle and completion times for groups of independent technological orderings.

which assumes that a subprogram is started as soon as possible, subject only to the "delays" that can occur due to the technological ordering. The idle time I_{mji} depends on the completion times of those subprograms which cover (mji). If there are several subprograms covering (mji), then the covering subprogram which takes the longest determines the idle time. We have

$$I_{mji} = \max \left\{ \begin{matrix} \max_{m'} T(m'ji' \rightarrow mji) - T(mji - 1) \\ 0 \end{matrix} \right\}; \tag{3.3}$$

i.e. idle time occurs when the maximum completion time for the subprograms which cover (mji) is larger than the time to complete the subprogram just prior to (mji) on the mth computer part.

In order to solve (3.1)–(3.3) for the idle and completion times in terms of the processing times we must specify those idle times which are given. We take those subprograms which do not cover any other subprograms (the starting subprograms) to have no idle time.

With these definitions and special values we can solve for the idle and completion times in terms of the processing times. We will simply illustrate the solution for the technological ordering given in Figure 2. The only subprogram not covering another subprogram is $(5j1)$. Hence

$$I_{5j1} = 0.$$

Thus from (3.1) we have

$$T(tj1) = t_{5j1}\,.$$

From (3.3) we have

$$I_{3j1} = \max \left\{ \begin{matrix} \max_{m'} T(m'ji' \rightarrow 3j1) - T(3j0) \\ 0 \end{matrix} \right\}.$$

Since there is no 0-th time on the third machine we take $T(3j0) = 0$ and the only subprogram covered by $(3j1)$ is $(5j1)$. Hence

$$I_{3j1} = t_{5j1}\,.$$

A similar argument holds for $(4j1)$:

$$I_{4j1} = t_{5j1}\,.$$

Then from (3.1):

$$T(3j1) = t_{3j1} + I_{3j1} = t_{3j1} + t_{5j1}$$

$$T(4j1) = t_{4j1} + t\,t_{5j1}$$

For $(2j1)$, we have from (3.1) that

$$T(2j1) = t_{2j1} + I_{2j1} ;$$

but from (3.3) we have:

$$\begin{aligned} I_{2j1} &= \max_{m'} \begin{Bmatrix} T(m'ji' \to 2j1) - T(2j0) \\ 0 \end{Bmatrix} \\ &= T(3j1) \\ &= t_{3j1} + t_{5j1} \end{aligned}$$

We can continue on in this manner and solve for all the completion times.

The idle time before $(5j3)$ is interesting, for there are three subprograms covering $(5j3)$. This idle time is:

$$\begin{aligned} I_{5j3} &= \max \left\{ \begin{matrix} \max \begin{Bmatrix} T(2j1) \\ T(4j2) \\ T(5j2) \end{Bmatrix} - T(5j2) \\ 0 \end{matrix} \right\} \\ &= \max \begin{Bmatrix} T(2j1) - T(5j2) \\ T(4j2) - T(5j2) \\ 0 \end{Bmatrix} \\ &= \max \begin{Bmatrix} t_{5j1} + t_{3j1} + t_{2j1} - (t_{5j1} + t_{4j1} + t_{5j2}) \\ t_{5j1} + t_{4j1} + t_{4j2} - (t_{5j1} + t_{4j1} + t_{5j2}) \\ 0 \end{Bmatrix} \end{aligned}$$

4. *The Many-Program Case.*

One of the aims of multiprogramming is to schedule a set of programs such that the total time to complete all the programs is a minimum. Finite problems of this type are extremely difficult and have been successfully solved in very special cases [7]. However, another approach to the problem has come to the fore in recent years: a technique based on digital simulation [3]. In simulation other mathematical *questions* are of relevance and so far seem to be answerable [3, 4]. Many of the relevant *questions* revolve about combinatorial *questions*; and these *questions* can be answered via a formulation which brings out the dependence of idle and completion times on the permutations of each subprogram through the computer parts. We proceed to formulate the sequencing questions of multiprogramming for a fixed set of independent programs, i.e. technological orderings.

If we had some multiprogramming schedule of a set of independent programs,

we could view a particular computer part and note which subprograms were processed sequentially in time. The collection of all computer orderings is called a schedule ordering. Without going into the mathematical details of formulating idle and completion times we will give a simple example of the concept of scheduled ordering and the choices of scheduled orderings which could minimize the total completion time of all subprograms.

In Figure 4 the linear graphs are given for two independent programs. We can construct any schedule provided the technological orderings implied by the linear graphs are preserved. For example, in Program 1 $(211 \leqslant (313)$ and in any schedule this relation must be preserved.

In Figure 5 two schedules are given from which it is easily checked that both sets of schedule order relations do not violate the technological order relations. This question of nonviolation of order relations is referred to as the consistency problem [cf. [4] for a discussion in the case of MSS]. The question of which schedule is better revolves around the desire to finish all programs in the shortest time.

When we are given the processing times t_{mj}, we can determine for any set of consistent schedules which schedule(s) gives the minimum time by simply drawing the Gantt diagrams and reading off the schedule times. In Figure 6 the Gantt diagrams are given for the linear graphs of Figure 5 and the Gantt diagram when each program is performed separately. For the processing times,

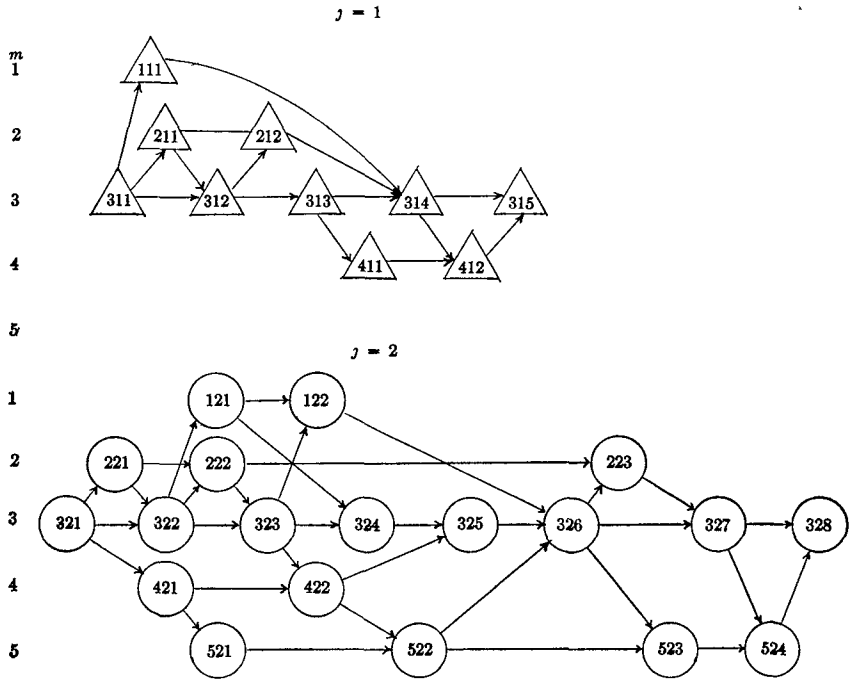

FIG. 4. Linear graphs for two independent programs

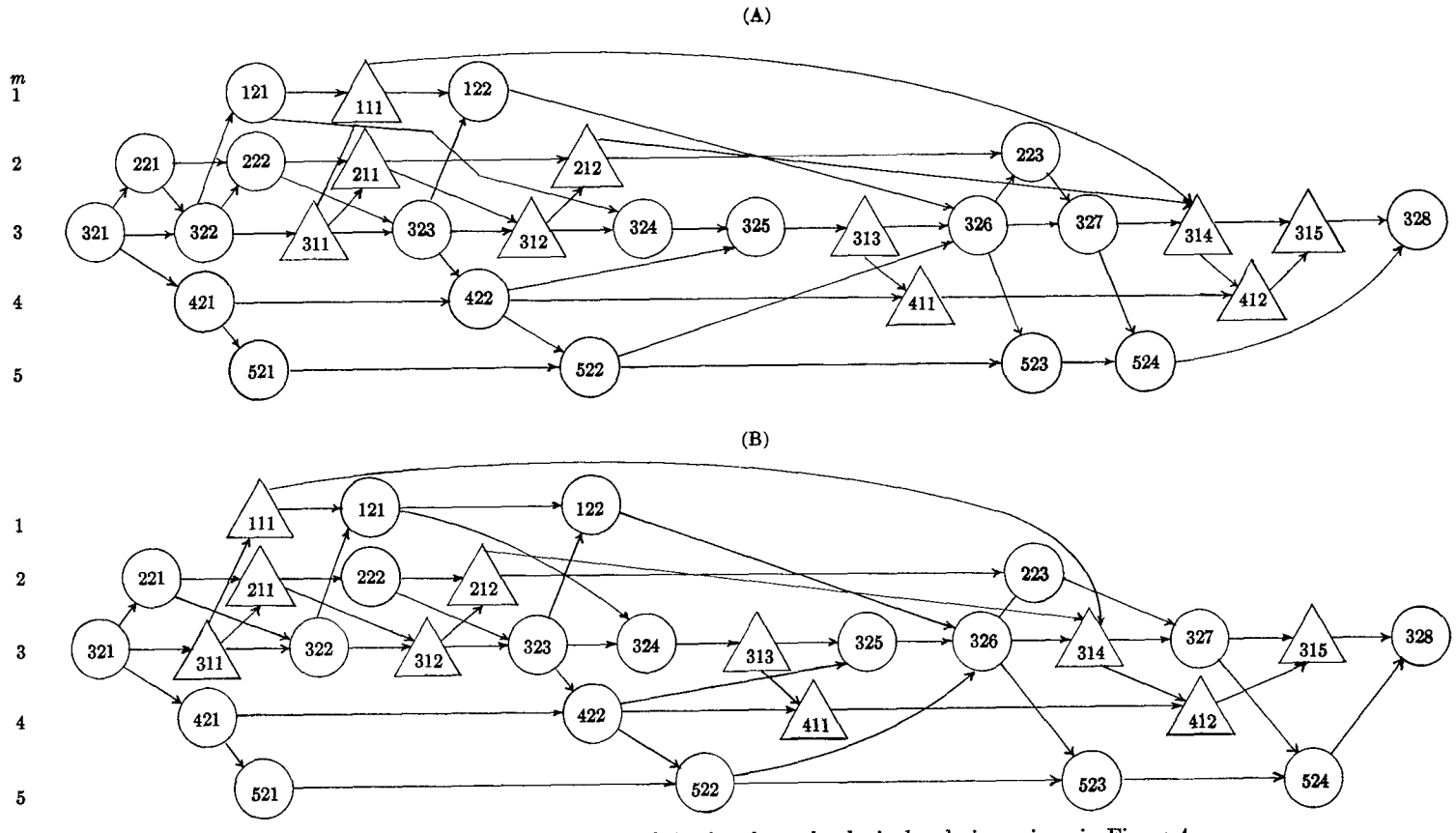

FIG. 5. Two possible consistent schedules for the technological orderings given in Figure 4

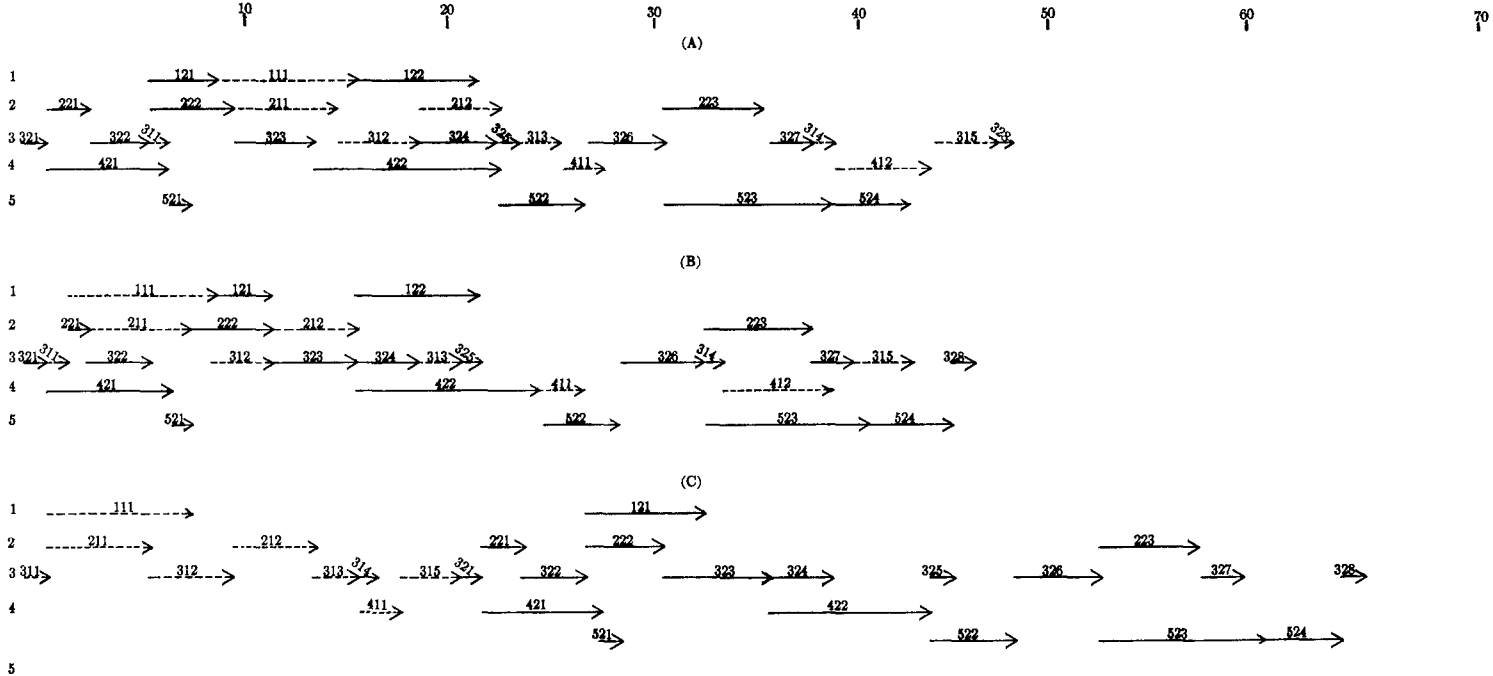

FIG. 6. Gantt diagrams for the schedules given in Figure 5. The last Gantt diagram is for each program done independently of the other.

TABLE I

$t_{111} = 7$									
$t_{211} = 5$	$t_{212} = 4$								
$t_{311} = 1$	$t_{312} = 4$	$t_{313} = 2$	$t_{314} = 1$	$t_{315} = 3$					
$t_{411} = 2$	$t_{412} = 5$								
$t_{121} = 3$	$t_{122} = 6$								
$t_{221} = 2$	$t_{222} = 4$	$t_{223} = 5$							
$t_{321} = 1$	$t_{322} = 3$	$t_{323} = 4$	$t_{324} = 4$	$t_{325} = 1$	$t_{326} = 4$	$t_{327} = 2$	$t_{328} = 1$		
$t_{421} = 6$	$t_{422} = 9$								
$t_{521} = 1$	$t_{522} = 4$	$t_{523} = 8$	$t_{524} = 4$						

t_{mji} are given in Table I. Both schedules are very much better than each program run separately, and there is not much difference in the schedule time between the two schedules.

We proceed to discuss the question of determining the processing times of the subprograms and the technique of scheduling a set of programs for a long production run on a computer.

5. *Some Technical Questions Involved in Multiprogram Scheduling*

Since a feasible approach to scheduling problems has appeared [5], we can look to the day when multiprogram scheduling can be performed and the *schedule executed* on a large scale digital computer. Various interesting questions arise when we attempt to multiprogram a set of programs: (i) How do we determine a subprogram size? (ii) How are the processing times of each subprogram determined? (iii) How is the computer used to multiprogram a set of programs and then execute this schedule? (iv) How do we determine before running a set of multiprogrammed schedules whether the capacity of the computer will be exceeded?

The question of subprogram size is difficult to determine at present for essentially no experience in multiprogramming is as yet available. Subprogram sizes would have to depend partly on the computer system and partly on the types of problems envisaged. There is some thought about the subprogram size for the IBM 709 input-output system [7], but as yet there is no general discussion of this problem.

Let us assume that some system of subprogram size is determined, i.e. the translation of an algebraic code will be performed if the program has indicated subprograms of a suitable size. The time to execute each subprogram (the processing time t_{mji}) will be of a stochastic nature: the subprogram times will depend on the amounts of data involved in input, output and processing, and even if the total amount of data involved were known, convergence rates of iterative processes would affect the subprogram time. The tendency to have algebraic translation systems with variable dimension statements may add to the difficulty in

determining subprogram times. Presumably some information will have to be supplied by the algebraic translation system for the running code which can be interpreted by a master program which computes the subprogram times for given data. The master program will then multiprogram a given set of programs. It is easy to state these desires, but we will have to expend great effort to achieve these desires.

There is one case in which subprogram times can be determined experimentally. In a data processing center the production runs are of known (experimental) length because of the repetitiveness of the work and the known amount of data used in each run. It is true that some variation in data occurs, but a simple argument using a Central Limit Theorem will show that a large variation from the expected processing time is very unlikely and hence it can be assumed that the subprogram processing time is the expected subprogram processing time.

(Since we can have idle time for each program separately, we may desire the algebraic translation system to choose a set of subprogram sizes for a program such that the total running time of this program is a minimum. We have not formulated this problem, similar to economic job lot sizes in MSS, because it adds complications to an already formulated complex problem. This problem, however, is worthy of study. How to "best schedule" one program and then multiprogram a set of "best scheduled" programs is not clear. Special cases could be studied from the point of view of decision theory under uncertainty.)

Suppose the very difficult questions of subprogram times have been overcome. What needs to be multiprogrammed? First, a program is needed to determine the multiprogram schedule; second, a program is needed which can direct the sequencing of the subprograms through each computer part.

The first question has been answered for the case of flow shop scheduling [5] and is believed applicable to all scheduling problems. The procedure is to sample schedules randomly and choose the best schedule after a number of trials. The number of trials is determined by decision theoretic methods. There is, however, one big difference: the computer which does the multiprogram scheduling also executes the programs. Just how to formulate this point needs careful consideration.

The question of traffic control of all the subprograms needs investigation. A program will have to be devised which retains control over the flow of subprograms. Presumably each subprogram on completion transfers control back to the master traffic control program.

The question of a computer's capacity to execute a set of programs simultaneously has received some attention. This problem is by no means understood and will require a great deal of study before it can be settled.

REFERENCES

1. BERGE, C. *Theorie des Graphes et Ses Applications*. Dunod, Paris (1958).
2. BIRKHOFF, G. *Lattice Theory*. Am. Math. Soc. Coll. Pub. XXV, New York City (1948).
3. HELLER, J. Combinatorial properties of machine shop scheduling. AEC Research and Development Report NYO-2879 (1959).

4. HELLER, J. Some numerical experiments for an $M \times J$ flow shop and its decision-theoretical aspects. *J. Op. Res. Soc. 8* (1960), 178–184.
5. MOCK, O., AND SWIFT, C. J. The Share 709 System: Programmed input-output buffering. *J. ACM 6* (1959), 145–151.
6. SISSON, R. L. Methods of sequencing in job shops—a review. *J. Op. Res. Am. 7* (1959), 10–29.
7. SISSON, R. L. Results of survey on sequencing theory. Draft Copy (Sept. 1959).