Discussed is the design, implementation, and detailed operation of an experimental IBM 7090 *program that calculates the critical path as well as early and late start times for a project network.*

The program accepts suitably coded information about activity durations and precedence relationships and constructs an internal representation of the network. Results are printed out in either one of two formats: one for the consistent case and the other for the inconsistent case.

A formal descriptive language is used to describe the basic algorithm. The specific storage and indexing techniques employed in the program are useful in a wide class of directed-graph applications.

High-speed calculation of the critical paths of large networks

by M. Montalbano

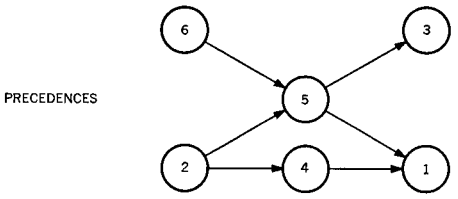
During an investigation of several project network techniques for their applicability to the effective description of business problems, it became evident that more efficient computer programs than those currently available could be written for large networks. Heretofore, critical-path computations have been divided into several phases. The experimental program described here employs a one-phase calculation procedure that eliminates much of the preprocessing of input data required by other methods and therefore achieves faster calculation speed. Written for the IBM 7090, the program calculates the critical path as well as early and late start times for large project networks without requiring the storage of intermediate results on magnetic tape. A formal descriptive language is used to describe the basic algorithm, first at a functional level and then at a machine-dependent level.

The network is symbolized by a directed graph whose nodes represent activities and whose arrows—also called *arcs*—represent precedence relationships between activities. The program accepts suitably coded information about activity durations and precedence relationships, constructs an internal representation of the network, and applies a calculation algorithm described more fully later.

assumptions and conventions

Many of the large PERT and CPM programs represent activities as arrows, and events as nodes. Such a representation differs from the one used in this paper where nodes represent activities, and an arrow connecting two nodes indicates that the activity at the tail

Figure 1 Methods of representing an activity network



$$P = \begin{pmatrix} 0&0&0&0&0&0 \\ 0&0&0&1&1&0 \\ 0&0&0&0&0&0 \\ 1&0&0&0&0&0 \\ 1&0&1&0&0&0 \\ 0&0&0&0&1&0 \end{pmatrix}$$

DURATIONS

NODE	1	2	3	4	5	6
TIME	15	5	30	20	10	7

$d = (15, 5, 30, 20, 10, 7)$

A. PRECEDENCE NETWORK AND TABLE OF DURATIONS

B. PRECEDENCE MATRIX AND DURATION VECTOR

of the arrow must be completed before the activity at the arrowhead can start. Dimsdale[1] discusses how to transform the event-node representation to the activity-node representation.

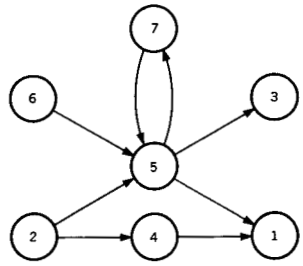
Figure 1 exemplifies the graphic and symbolic conventions customarily employed to describe activity-node networks: Figure 1A displays a directed graph and an associated table which gives duration times for each activity; Figure 1B represents the same network by means of a precedence matrix, P, and a duration vector, d. Both of these representations are used in this paper.

The major restrictions limiting the generality of CPM002, the program described here, are concerned with numbering nodes and representing times. Activity (node) numbers must be specified as positive integers requiring no more than four decimal digits in their representation (the upper limit on node numbers depends on memory allocation); times must be specified as integers in the range 0 to 32,767 inclusive. The program does not calculate times greater than 32,767. This restriction does not seem unduly stringent: a network whose activities are measured in calendar days can have a maximum duration time of approximately ninety years; in work days, the maximum time possible is, of course, even greater.

Networks in which activity numbers and times are not represented by integers in the appropriate range must have, in addition to the single-pass calculation described below, two additional passes to code and decode the original representations of activities and times into the representations required by the program.

In discussing the figures in this paper, the following conventions are used: a node with no arrows pointing to it, or with only dashed arrows pointing to it, is called a *source;* a node with no arrows pointing away from it, or with only dashed arrows pointing away from it, is called a *sink;* the calculation consists of an iteration of successive steps that associate time values (represented by numbers) with sources and sinks; the time values for sources are called *forward times*, those for sinks *backward times;* those sources for which forward times have not been calculated at the start of

Figure 2 Graph of an inconsistent precedence network



PRECEDENCES

$$P = \begin{pmatrix} 0&0&0&0&0&0&0 \\ 0&0&0&1&1&0&0 \\ 0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0 \\ 1&0&1&0&0&0&1 \\ 0&0&0&0&1&0&0 \\ 0&0&0&0&1&0&0 \end{pmatrix}$$

DURATIONS

NODE	1	2	3	4	5	6	7
TIME	15	5	30	20	10	7	22

$d = (15, 5, 30, 20, 10, 7, 22)$

A. PRECEDENCE NETWORK AND TABLE OF DURATIONS

B. PRECEDENCE MATRIX AND DURATION VECTOR

an iteration are called new sources; in contrast, those for which times have been calculated are called old sources; similarly, we speak of new and old sinks determined by the calculation of backward times.

The graph described in Figure 1 is *consistent* because it has no sequence of arrows starting and ending at the same node. The graph described in Figure 2 is *inconsistent* since one arrow leads from Node 5 to Node 7 and another one from Node 7 back to Node 5. Critical paths can be calculated only for consistent networks; thus, a computer program to calculate critical paths must include some provision for consistency checking.

When node numbers are arranged in a serial list such that each node occurs earlier in the list than any of its successors, the nodes are said to be arranged in *topological order*. Many critical-path programs facilitate the calculation of forward and backward times by arranging the nodes in a topologically ordered list before doing the time calculations. This is frequently called *topological sorting*.

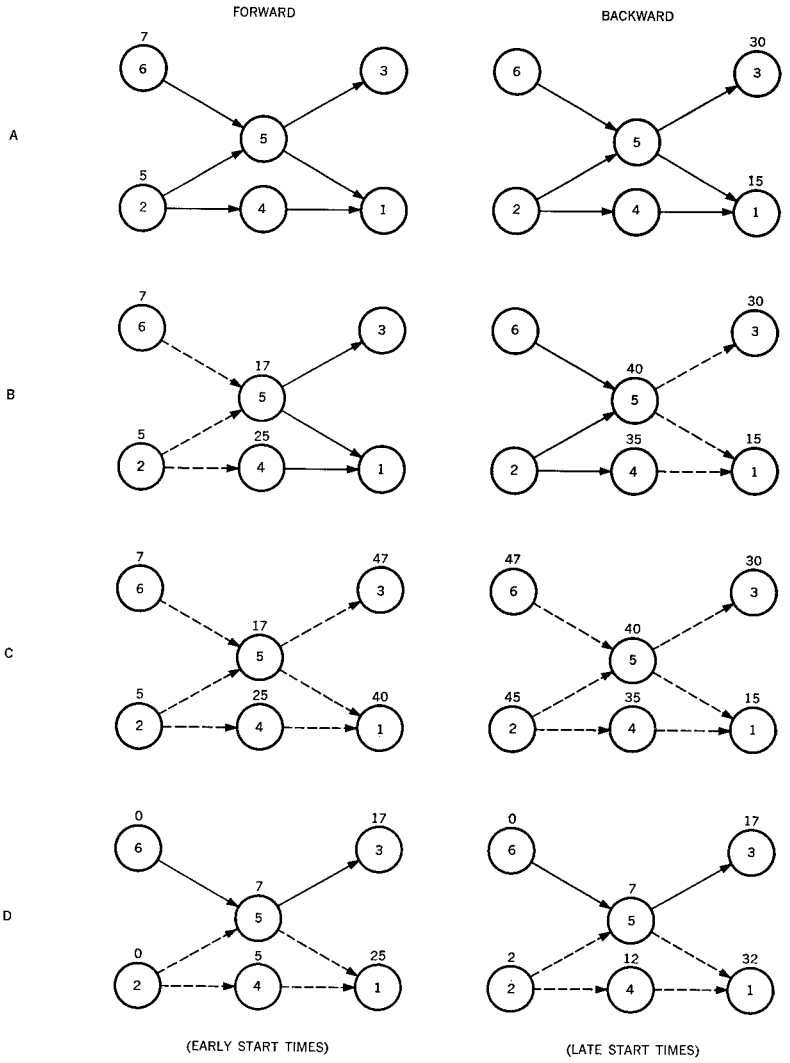
program features

In calculating early and late start times for activities, most programs have a forward time calculation phase followed by a backward time calculation phase. For large networks, both of these phases are frequently separate tape passes, as are the consistency-checking and topological-sorting phases described above.

The main features of CPM002 are:

- Consistency checking is a by-product of the basic procedure for calculating forward and backward times.
- Topological sorting is not required.
- Forward and backward time calculations are overlapped—i.e., done concurrently—in a manner described more fully below.
- Information describing the precedence matrix and duration vector is stored in the IBM 7090 computer in a manner that makes it possible to calculate relatively large networks without writing intermediate results on tape.

Figure 3 Graphic representation of steps in algorithm for consistent network

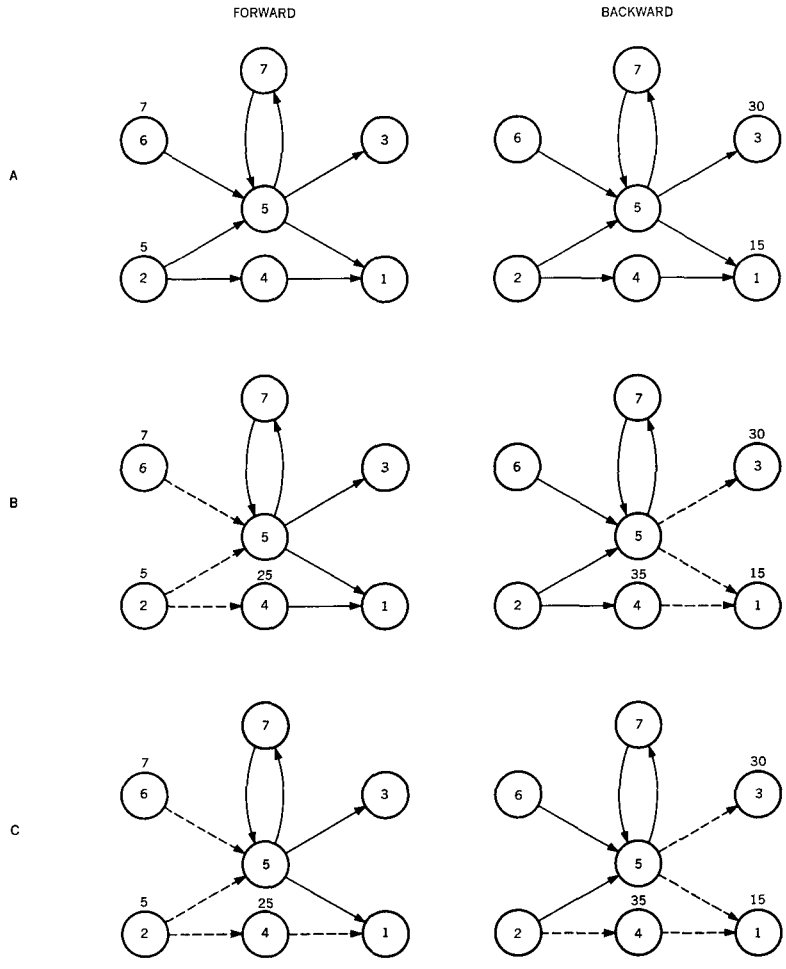


- The organization of the program as a set of independent subroutines working on the same arrays permits easy extension of the same techniques to other problems in graph and network analysis, such as calculating shortest distance rather than longest time, eliminating redundant precedence relations, determining graph isomorphisms, etc.

Description of basic algorithm

The description of simple procedures like those of CPM002 ordinarily requires an unsatisfactory, verbose, vague combination of words and pictures. For this reason, the narrative description of the basic algorithm is followed in this paper by the Iverson-language

Figure 4 Graphic representation of steps in algorithm for inconsistent network



description. The latter program is included both in its own right (as the most concise, precise, and useful description the author can provide of the basic calculations carried out by CPM002) and as an indication of the nature of the research from which the program derives. The program used here is an early form of the language now implemented as the APL\360 time-sharing system.

narrative description

Figure 3 depicts successive stages in the basic algorithm used in CPM002, as they would occur in processing the consistent network described in Figure 1. Figure 4 depicts the stages that occur in processing the inconsistent network of Figure 2.

The earliest time at which an activity can start is the time at which the latest of its predecessors is finished. The latest time at which an activity can finish is the time at which the earliest of its successors must start. This symmetry can be used to calculate backward times in a manner exactly analogous to calculating forward times and concurrently with the calculation of forward times,

i.e., without a prior determination of the critical-path or minimum project duration time. Thus, the steps in Figures 3A, B, and C are accomplished by CPM002 in three scans of the precedence matrix rather than in the six scans that would be required if forward times were calculated before backward times. It is important to keep this point in mind throughout the following discussion.

The steps in the basic algorithm are described in terms of operations on nodes and arrows.

Step 1. Locate new sources and new sinks. If there are none, the algorithm is at an end. The ending procedure is described in Step 5. (Note the parallel treatment of forward and backward calculations in Figures 3 and 4.)

Step 2. Calculate forward times for new sources and backward times for new sinks. A forward time is calculated for each new source as the sum of its duration time plus the forward time of the one or more of its predecessors, if any, whose forward time is greatest. (Note that predecessors are connected to new sources by dashed arrows pointing to the new source.) A backward time is calculated for each new sink as the sum of its duration time plus the backward time of the one or more of its successors, if any, whose backward time is greatest. (Note that successors are connected to new sinks by dashed arrows pointing away from the new sink.)

Step 3. Replace, by dashed arrows, all the solid arrows pointing away from the new sources that have just been processed. Replace, by dashed arrows, all the solid arrows pointing to the new sinks that have just been processed. (For consistent networks, this either develops new sources and sinks for the next iteration or, if no solid arrows remain, indicates that the calculations are complete.)

Step 4. Repeat Step 1.

Step 5. This is the ending procedure. If all the nodes in the network have both forward and backward times associated with them, the network is consistent. In this case, an early start time can be calculated for each node by subtracting its duration time from its forward time; a late start time can be calculated for each node by subtracting its backward time from the largest backward (or forward) time found in the network. This largest time is the critical-path or minimum project duration time. Those nodes whose early and late start times are equal lie on the critical path.

If all the nodes do not have both forward and backward times, the network is inconsistent (as described later).

Let us first discuss the consistent case. Figures 3A, B, and C depict the three iterations required by the basic algorithm to calculate forward and backward times. These iterations consist of repetitions of the above Steps 1 through 4. Figure 3D depicts the calculation of early and late start times after the basic algorithm is completed.

Iteration A. The new sources (in this case, the original sources) are Nodes 2 and 6. The new sinks are Nodes 1 and 3. Since the new sources have no predecessors and the new sinks have no successors, the forward and backward times are merely the duration times of each activity. These are written above their corresponding nodes, as are all the times calculated in subsequent iterations.

Iteration B. New sources: Nodes 4 and 5. New sinks: Nodes 4 and 5. Forward times: Node 4, predecessor time plus duration time: $5 + 20 = 25$; Node 5, greatest predecessor time plus duration time: $7 + 10 = 17$. Backward times: Node 4, successor time plus duration time: $15 + 20 = 35$; Node 5, greatest successor time plus duration time: $30 + 10 = 40$.

Iteration C. New sources: Nodes 1 and 3. New sinks: Nodes 2 and 6. Forward times: Node 1, greatest predecessor time plus duration time: $25 + 15 = 40$; Node 3, predecessor time plus duration time: $17 + 30 = 47$. Backward times: Node 2, greatest successor time plus duration time: $40 + 5 = 45$; Node 6, successor time plus duration time: $40 + 7 = 47$. Since no new sources or sinks are developed by iteration C, no further iterations are needed. The process of calculating early and late start times ends with Step 5.

Step 5. Early start times are calculated by subtracting activity durations from the forward times. Late start times are calculated by subtracting the backward times from the project duration time (in this case 47—the longest time associated with any node either as a forward or a backward time). The critical path (indicated by solid arrows in Figure 3D) is made up of those nodes for which early and late start times are equal.

For the inconsistent case, the network in Figure 4 is basically the same as that of Figure 3, except that an extra node (7) and two arrows have been added in such a way as to introduce inconsistent precedence relationships, called a *circuit.* As a result, the process again terminates after three iterations (since no new sources or sinks can be found), but this time without having calculated both forward and backward times for all nodes or—equivalently—without converting all the solid arrows in both diagrams to dashed arrows. (The reason for the termination at iteration C is evident if the definitions of new sources and new sinks are recalled.)

The information displayed in the two graphs of Figure 4C permits us to determine that the network contains an inconsistency (i.e., a circuit) and to divide all the nodes in the network into four groups:

(1) Nodes with both predecessors and successors in the circuit (in this case, 5 and 7)
(2) Nodes with successors in the circuit (2 and 6)
(3) Nodes with predecessors in the circuit (1 and 3)
(4) Nodes with neither successors nor predecessors in the circuit (4)

These are respectively: (1) nodes for which we have calculated neither forward nor backward times; (2) nodes for which we have calculated forward but not backward times; (3) nodes for which we have calculated backward but not forward times; and (4) nodes for which we have calculated both forward and backward times.

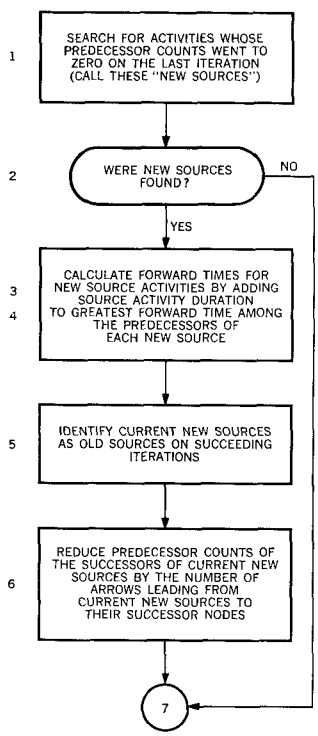
The information given by this subdivision can clearly be of great value in locating the error or errors that caused the inconsistency.

Iverson-language description

Narrative descriptions are, at best, merely general indications of how a program works. Converting the thinking about nodes and arrows into computer program steps that manipulate bits, words, and arrays frequently requires as much hard work as devising the node-arrow descriptions in the first place. It is also frequently hard to demonstrate the connection between the two. The same objection applies to narrative flowcharts of the kind displayed in Figure 5 which describes the first six steps of the Iverson program shown in Figure 6.

A properly constructed Iverson-language[2,3] algorithm, on the other hand, not only describes what has to be done concisely, precisely, and vividly, it also guides efficient programming design.[4] This is done in Figure 6. A brief discussion of this figure and its relation to CPM002 is now offered.

Figure 5 Conventional flowchart corresponding to first six steps of Iverson-language program



Steps 1 through 6 of Figure 6 describe the processing for the forward-time calculation: Steps 7 through 12 describe exactly analogous processing for the backward-time calculation; Step 13 is merely a repetition of Step 2. This is required by the condition that the algorithm stop only when both the conditions $\vee/r = 0$ and $\vee/u = 0$ are met, since we are interested in providing information about circuits. (Only one of these three comparisons would be needed if we knew the network was consistent since $\vee/r$ would then attain the value 0 at the same iteration as $\vee/u$.)

Since the basic ideas can be illustrated in terms of Steps 1 through 6, the following discussion concentrates on these steps. The basic entities involved are:

1. The precedence matrix, P (see Figure 1)
2. The duration vector, d (see Figure 1)
3. The predecessor vector, p

$$p = +//P, \quad \text{that is,} \quad p_j = \sum_{i=1}^{N} P_{ij}$$

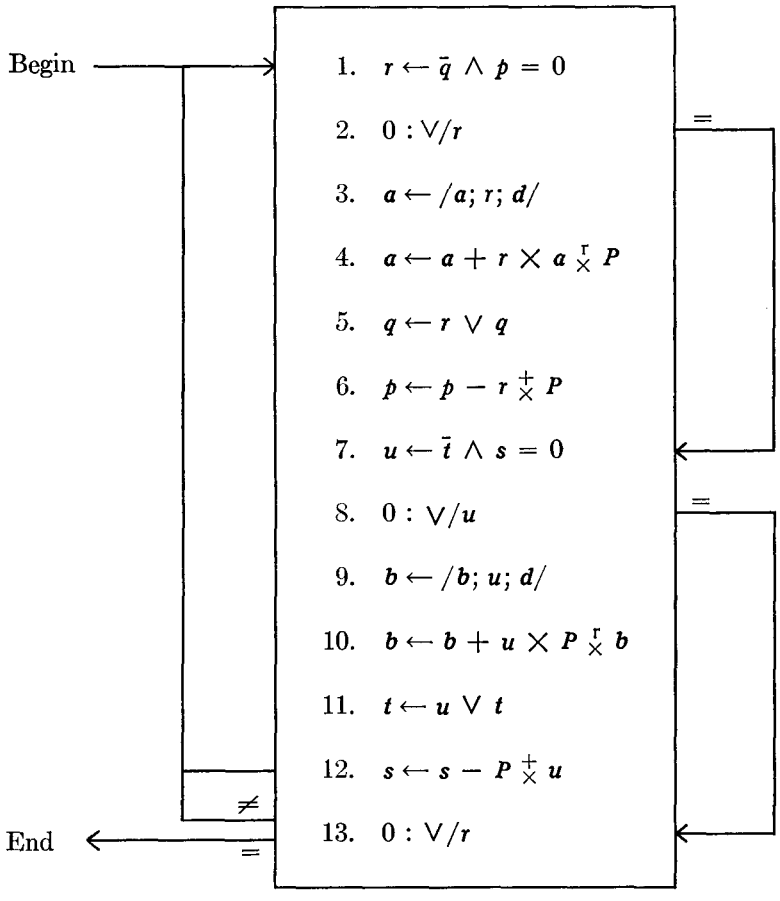
where N is the number of nodes in the network

4. The new-origin vector, r
5. The old-origin vector, q (initialized so that its components are all zero prior to execution of the algorithm)
6. The forward-time vector, a

(In the backward-time calculation, steps 7 through 12, the vectors analogous to p, r, q, and a are s, u, t, and b.)

P is a logical matrix, i.e., it has components that take on only 0 or 1 as values. The other entities are vectors: logical in the case

Figure 6 Iverson-language description of steps in algorithm



of r and q; numerical in the case of a, d, and p. All the vectors have N components; P has $N \times N$ components.

Step 1. Calculate r, the new origin vector. Its components are 0, except in positions corresponding to the occurrence of 0's in both the old-origin and predecessor vectors; in these positions, the components of r have the value 1.

Step 2. Test to see if the new-origin vector has at least one nonzero component. If it has not, no further processing in the forward direction is possible and the forward calculation is suspended.

Step 3. Copy into vector a the components of d, the duration vector that corresponds in position to 1's in the new-origin vector.

Step 4. Multiply the components of each column vector in P by the corresponding components of a. Form a result vector consisting of the maximum component in each column. Add to corresponding components of a those components of this result vector that correspond in position to 1's in the new-origin vector, r.

Step 5. Convert from 0 to 1 those components of the old-origin vector, q, that correspond in position to 1's in the new-origin vector, r.

Step 6. Multiply P by r and subtract the result vector, component by component, from the predecessor vector, p.

The steps constitute a formal, functional description of the basic algorithm underlying CPM002. This functional description was used as a guide to more detailed Iverson-language programs. In these programs, the basic functions were realized by procedures that attempted to capitalize on the equipment features of the IBM 7090 and the operating system features of the IBSYS/IBJOB compiler, assembler, and loader.

The main equipment features of the 7090 relevant to the program were word and register sizes and the operations affecting partial words and bits. These equipment features permit a 36-bit configuration to be considered either as a scalar (i.e., a vector with only one component), a four-component vector (prefix, decrement, tag, address), a 36-component vector, or a combination of these.

An examination of the Iverson program shows that we have two variable numerical vectors, a and p, and that non-zero values for the first are defined only when the second has gone to zero. Since both have the same number of components, this indicates that the same storage locations can be used to store both of these vectors. This, in fact, has been done. Predecessor counts are stored in the decrement portion of a series of 7090 words. As soon as the predecessor counts have gone to zero, the same decrement portions are used to store corresponding values of the forward-time vector, a.

The remaining variable vectors are *logical*, i.e. they take on component values requiring only one bit of storage. Since they are functionally related to the p and a vectors (i.e., they have the same number of components, and corresponding index values specify related status and numerical information), they can be thought of as vectors of flag bits. This is the way they are treated. The first bit of a 7090 word stores a component of the r vector; the second bit stores a component of the q vector.

The 7090 permits fixed-point operations in several registers. Among these, the accumulator register and the three index registers are particularly important. The index registers are fifteen bits long. If we can restrict the maximum result of an arithmetic operation to 32,767, we can do arithmetic, comparison, and information manipulation much more efficiently than by doing our arithmetic only in the accumulator. If we examine the characteristics of the class of problem we are trying to solve, we see that, by its very nature, the components of the p vector must be less than 32,767, since we are talking about networks with fewer than 10,000 nodes. We then ask whether we can restrict the values of the a vector to lie in the range 0 through 32,767 without loss of generality.

As we have pointed out previously, this range permits us to schedule a 90-year project as a set of subprojects whose durations

are specified in terms of days. Thus, the accuracy we can attain is much greater than inevitable errors in our time estimates and, for all practical purposes, the restriction of time values to 32,767 time units is certain to give us all the accuracy we can use.

The question of calendar conversion, of course, remains; but it would do so in any case. Calculation from calendar days to work days and vice versa is a fairly straightforward operation which can be done in a variety of ways.

From the symmetry of our approach to the forward- and backward-time calculation problem, we see that—once we have determined that we can store all the forward-time information we need in seventeen bits of a 7090 word—we can store all the backward-time information in another seventeen bits. The array C, discussed later, can be thought of as a collection of N 7090 words, each word being treated as if it had eight components. Conceptually, this array is a matrix C of dimension $N \times 8$, and each of its columns stores the following variable of our functional description:

C_1	:	r	1-bit components
C_2	:	q	
C_3	:	unused	
C_4	:	p; a	15-bit component
C_5	:	u	1-bit components
C_6	:	t	
C_7	:	unused	
C_8	:	s; b	15-bit component

Having selected the way of storing the variable information, the major design decision of how to store the constant information (precedence matrix, P, and duration vector, d) remains.

A first temptation is to store P as an array of bits. For networks of the size we are considering, however, this is clearly impractical. A 3000-node network would require 9,000,000 bits or 250,000 words. This is inconsistent with our objective to design a program that can process large networks entirely in the 32,768 words of high-speed storage available on the 7090.

Here again, a consideration of the characteristics of real-life problems is required. Precedence matrices describing real problems are customarily extremely sparse, i.e., they have few non-zero elements. Assuming, for example, that we have an average of ten non-zero elements per row, information about 30,000 components must be stored in the case of a network of 3000 nodes; these components would have the value 1 in the conventional representation of P, all others would have the value 0. Thus, we can specify P as the set of 30,000 (i, j) values specifying its non-zero components.

A consideration of 7090 specifics is helpful here. If we store these (i, j) values as a row list, the i value can be calculated rather than stored; all we need do is store sequences of j values and indicate when we are storing either the first or last non-zero j of a row. We have thus reduced our storage requirements from 30,000 (i, j) pairs to 30,000 j values and 3000 flag bits. Since the j values can be

Table 1 CPM003 and CPM004 subroutines and control cards

CPM003	CPM004
CPMOVR	CPMOVR
A3000	A1000
B165C	B5000
STORNJ	STORNJ
$ORIGIN ABLE	
$INCLUDE (Various I/O subroutines)	
$ORIGIN CHAS	
RDCPIO	RDCPIO
FRDCPM	FRDCPM
$ORIGIN CHAS	
CPMOUT	CPMOUT
WRCPM	WRCPM
$ORIGIN CHAS	
WRBCKT	WRBCKT
FWCT	FWCT
$ORIGIN ABLE	
C3000	C1000
$ORIGIN BAKER	
PSCT	PSCT
$ORIGIN BAKER	
LGTM	LGTM
$ORIGIN BAKER	
CCP	CCP
$ORIGIN BAKER	
CTOB	CTOB

stored in fifteen bits, the specific P matrix under discussion can be stored in 15,000 words. This is done in CPM002.

Program CPM002

Program CPM002 is a generic designation for a set of complementary subroutines and a main program that calls these subroutines in the order required to read, calculate, and write results for a batch of critical-path jobs. Three of these subroutines are actually BLOCK DATA subprograms that reserve space for arrays A, B, and C; A and B store the precedence matrix and the duration vector, whereas C stores all the variable information (vectors a, p, q, r, b, s, t, u). CPM002 can be tailored to specific requirements in two ways:

- The dimensions of the maximum basic arrays can be altered by rewriting the BLOCK DATA subprograms.
- $ORIGIN and $INCLUDE control cards can be used to overlay the area used by the read and write routines with the area used by the C array and the subroutines doing the critical-path calculations. This need be done only for networks requiring the extra storage space for arrays.

Two particular versions of CPM002 have been standardized as CPM003 and CPM004. Their subroutines and control cards are listed

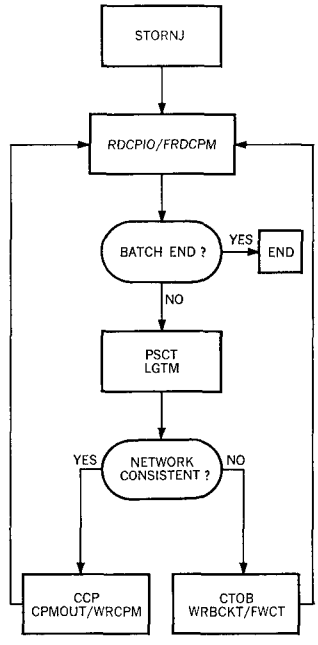
in Table 1. CPM003 can process any network of up to 3000 nodes and 33,000 arrows. CPM004 can process any network of up to 1000 nodes and 10,000 arrows. As can be seen from the listing, the only difference in the two programs is in the designation and contents of the BLOCK DATA subprograms (A3000, B165C, C3000 for CPM003; A1000, B5000, C1000 for CPM004) and in the control cards used for overlay purposes in CPM003.

basic subroutines

The main program is CPMOVR, written in FORTRAN. This program calls in subroutines in the sequence described by the flowchart of Figure 7. Briefly, the subroutines perform the following functions:

1. STORNJ is a MAP (Macro-Assembly Program) subroutine that provides heading information (including the date on which the calculations are performed) common to all jobs.
2. RDCPIO/FRDCPM is a pair of subroutines of which the first is written in MAP and the second in FORTRAN. Their joint function is to read and list all the input required to define one critical-path job and to store this information in the A and B arrays. If the input information is incorrect, the message INCORRECT DECK SETUP CALCULATIONS OMITTED is printed and succeeding cards are merely listed until a card starting a new job is encountered. These programs read cards whose functions are specified by a code in column 4. Code 9 identifies an end-of-batch card. Before RDCPIO/FRDCPM returns control to the main program, it sets a variable L to zero if arrays A and B have been properly stored, and to some nonzero value if it has encountered an end-of-batch card. The main program tests this variable to determine whether or not to continue processing.
3. PSCT is a MAP subroutine that counts predecessors and successors and stores these counts in the C array. These counts constitute the vectors $\boldsymbol{p}$ and $\boldsymbol{s}$ of the Iverson Program.
4. LGTM is a MAP subroutine that carries out the basic algorithm. If the network is consistent, a variable I is set to zero. If the network is not consistent, I is set to some nonzero value. The main routine calls CCP if I is zero, and CTOB if it is not.
5. CCP is a MAP subroutine that calculates early and late start times. To permit overlay, CCP also copies this information from the C array to the B array. In other words, the B array information is lost when subroutine CCP is called.
6. CPMOUT/WRCPM is a MAP-FORTRAN pair of subroutines that calculates times and floats for a consistent network and prints them out.
7. CTOB is a MAP subroutine that merely copies the information stored in the C array into corresponding relative positions in the B array. Like subroutine CCP, its purpose is to permit overlaying the C array with input/output subroutines. Note that the B array must be of a size greater than or equal to the C array.
8. WRBCKT/FWCT is a MAP-FORTRAN pair of subroutines that prints out information about inconsistent networks.

Figure 7 Subroutine sequence



input cards

Input cards are identified as to function and format by one of the following code numbers in card column 4: 1, 2, 3, 4, 9. Cards not having one of these codes in column 4 are listed as input but otherwise ignored. Comments can thus be interspersed anywhere in a job or batch. For all card types, the contents of columns 1 through 3 are listed but otherwise ignored. The function of columns 5 through 80 for each of the card types is now discussed.

Card Code 1

Format. Columns 6–11, job identification code (6 alphameric characters). Columns 13–24, requester or programmer name (12 alphameric characters). Columns 25–28, number of nodes in network (up to 4 numeric characters). The job identification code and the requester name are printed as part of the heading for the output listing. The job number is also printed as part of the end-of-job identification. Except for the heading and end-of-job information, the number of lines of output depends on the number entered in columns 25–28.

Function. Card type 1 is a start-job card. When it is encountered, all arrays are initialized to zero values, and the number-of-nodes information is stored to be used as a check that node numbers specified on succeeding cards do not exceed the maximum number specified by the first card. This number itself is checked to see that it does not exceed the dimension of the A and C arrays. If it does not, it is used as the working dimension of the A and C arrays. If it does, an error notice is printed and all subsequent cards are merely read and listed until either another number 1 or a number 9 is encountered.

Card Code 2

Format. Columns 5–8, node number (up to 4 numeric characters). Columns 9–12, activity duration (up to 4 numeric characters).

Function. Card type 2 is a node (or duration) card. The number found in columns 9–12 is entered in the decrement portion of a word in the A array. The relative location of the word in the A array is given by the node number found in columns 5–8.

Note. If more than one duration card is entered for a given node, the duration specified by the last one is the one that will be used by the program.

Card Code 3

Format. Columns 5–8, node number of predecessor node, i. Columns 9–12, 13–16, . . . 77–80, node numbers of successor nodes, j.

Function. Card type 3 is an arrow (or precedence) card. The i value (specified by the number in columns 5–9) is used to locate the word in the A array in which a pointer is to be stored. The j values (specified by the remaining numbers on the card) are stored in the next available positions in the B array. These numbers are stored in the decrement and address portions of

successive words in B. The last j value is identified by a flag bit. The pointer is stored in the A word and consists of an address and a flag bit that indicates whether the first j value is to be found in the decrement or address portion of the word at that address.

Note. If a node has more than 18 successors, they may be entered on as many more type-3 cards as needed. All of these cards must be entered as one group, however, i.e., no type-3 cards with a different i value may be interposed between cards of a like i value. In other words, if a group of cards identifying all the successors of Node i were to be separated into two subgroups by a type-3 card for Node k, for example, only the information about the second subgroup would be used by the program. This is the only sequencing restriction on type-2 or type-3 cards. Except for this, they can be entered in any order, provided they occur after an appropriate type-1 card and before an appropriate type-4 card.

Card Code 4

Format. Except for column 4, the information on a type-4 card is listed and ignored.

Function. Card type 4 is an end-of-job card. It signals that input for a job is complete and processing should start.

Card Code 9

Format. Except for column 4, the information on a type-9 card is listed and ignored.

Function. Card type 9 is an end-of-batch card. It signals that there are no more CPM jobs to be processed.

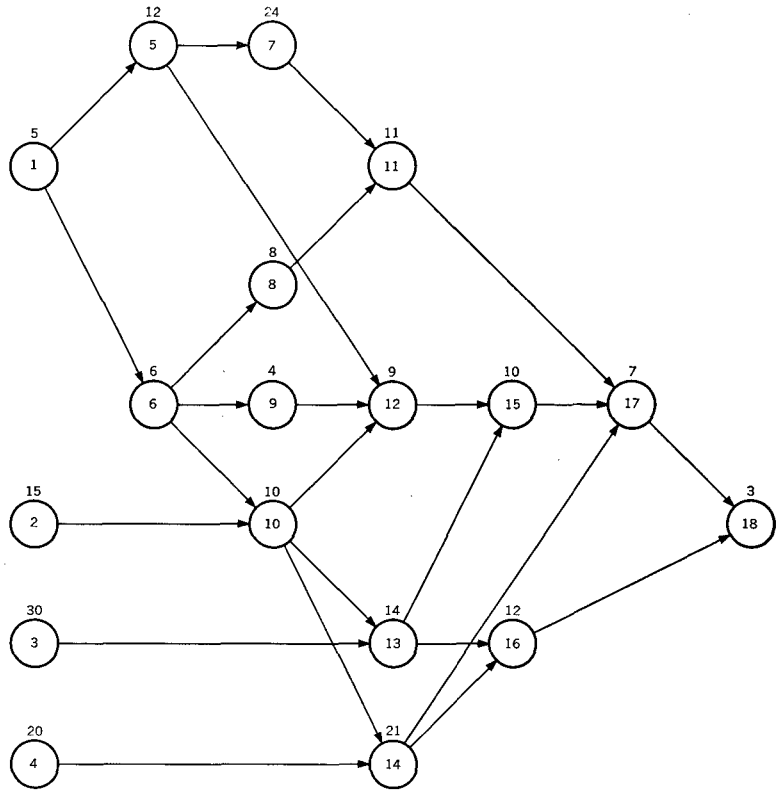
If any of the node numbers or times are less than four digits, they may be entered anywhere in the four-digit field. In particular, they may be left-justified rather than right-justified. However, no intervening blanks may separate the digits making up a number. There are a variety of checks on various parts of the input process. All precedence cards, for example, must have an i value in columns 5–8 and a j value in columns 9–12. (Other j values, however, may be entered in any of the remaining 17 fields.) There is only one error message, but it is issued immediately after the listing of the card on which the error was detected. Succeeding cards on the same job are not checked.

Except for the fields described above, alphabetic or numeric information may be entered anywhere on any of the input cards. This includes all the j fields, except the first one on the type-3 card, that is, all the fields after column 13, as long as the information in any field cannot be interpreted as a decimal number of from one to four digits. If only alphabetic comments are written to the right of the last successor value on a type-3 card, they are ignored. This is also true for most other combinations of letters and numbers.

The BLOCK DATA subprograms reserve space for the maximum A, B, and C arrays that are needed for a particular batch of jobs.

block data

Figure 8 Consistent network NETA[5]



If an array is to store N values, the dimension of the array is $N + 1$, and the first word is used to store the integral value N so that subroutines can determine when array bounds are being exceeded. The array A stored by subprogram A3000 in the CPM003 package, for example, reserves 3001 words and uses the first to store the number 3000, giving the effective, usable dimension of the array.

As examples, the actual FORTRAN subprograms defining the arrays used in CPM003 are given in Table 2. The dimensions of A and C are determined by the maximum number of nodes to be considered. The dimension of B is determined by the maximum number of arrows. A and C each require one word per node. B requires a half word per arrow. The deck names for the subprograms are chosen to indicate the dimensions of the arrays they specify. Since the deck names are not referred to by any program in any CPM002 package, none of the functional programs need be recompiled and the binary decks can be used without alteration.

program operation

The operation of a CPM002 program is now illustrated. A consistent network NETA is shown in Figure 8, and an inconsistent network CKTA in Figure 9. The networks are identical except for the dashed arrow from 11 to 6 and the three disjoint nodes 19, 20, and

Table 2 FORTRAN Block Data subprograms

```
$IBFTC  A3000
        BLOCK DATA
        INTEGER A(3001)
        COMMON/A/A
        DATA A(1)/3000/
        END
$IBFTC  B165C
        BLOCK DATA
        INTEGER B(16501)
        COMMON/B/B
        DATA B(1)/16500/
        END
$IBFTC  C3000
        BLOCK DATA
        INTEGER C(3001)
        COMMON/C/C
        DATA C(1)/3000/
        END
```

Figure 9 Inconsistent network CKTA

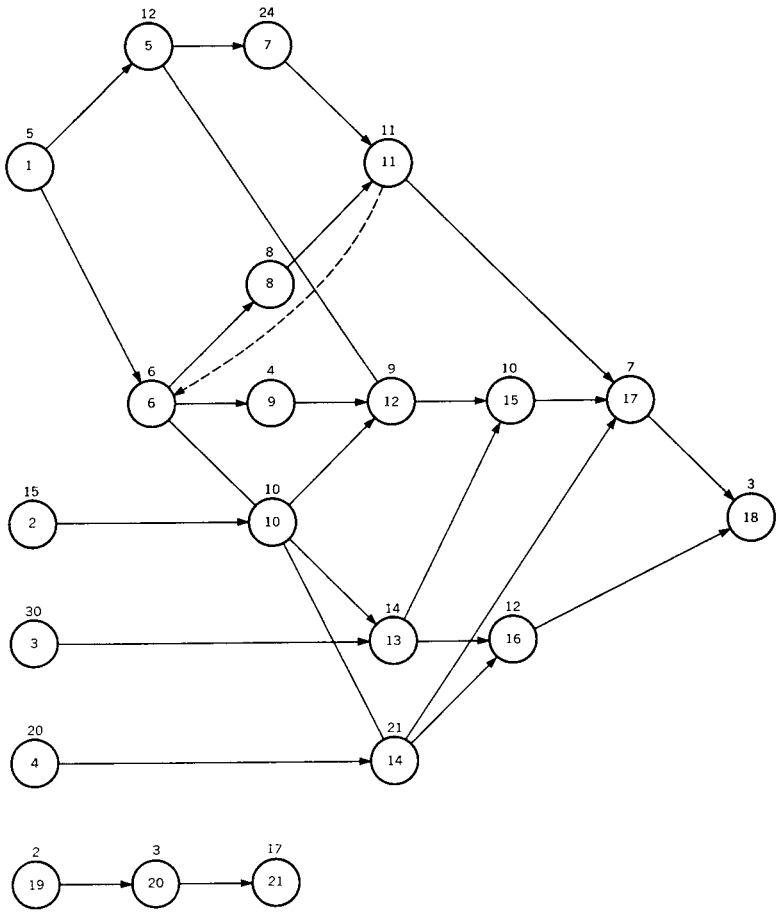


Table 3 Network NETA output

MONTALBANO JOB NETA DATE 101165 CPM002
MINIMUM PROJECT TIME 64

Node	*Durat.*	*EStart*	*E End*	*LStart*	*L End*	*TFloat*	*FFloat*	*Node*
1	5	0	5	2	7	2	0	1
2	15	0	15	3	18	3	0	2
3	30	0	30	******	******	******	******	3
4	20	0	20	8	28	8	5	4
5	12	5	17	7	19	2	0	5
6	6	5	11	12	18	7	0	6
7	24	17	41	19	43	2	0	7
8	8	11	19	35	43	24	22	8
9	4	11	15	31	35	20	10	9
10	10	15	25	18	28	3	0	10
11	11	41	52	43	54	2	2	11
12	9	25	34	35	44	10	10	12
13	14	30	44	******	******	******	******	13
14	21	25	46	28	49	3	0	14
15	10	44	54	******	******	******	******	15
16	12	46	58	49	61	3	3	16
17	7	54	61	******	******	******	******	17
18	3	61	64	******	******	******	******	18

END OF JOB NETA

Table 4 Network CKTA output

MONTALBANO JOB CKTA DATE 101165 CPM002
INCONSISTENT NETWORK

	Circuit Nodes		*Entry Nodes*		*Exit Nodes*	
Node	*Pred.*	*Succ.*	*Time*	*Succ.*	*Pred.*	*Time*
1			5	2		
2	*****					
3	*****					
4	*****					
5			17	1		
6	1	1				
7			41	1		
8	1	1				
9					1	33
10					1	46
11	1	1				
12					2	29
13					1	34
14					1	36
15					2	20
16					2	15
17					3	10
18					2	3
19	*****					
20	*****					
21	*****					

Table 5 Network NETA input

CPM 1	NETA	MONTALBANO		18	TEST NETWORK A
CPM 2	1	5			
CPM 2	2	15			
CPM 2	3	30			
CPM 2	4	20			
CPM 2	5	12			
CPM 2	6	6			
CPM 2	7	24			
CPM 2	8	8			
CPM 2	9	4			
CPM 2	10	10			
CPM 2	11	11			
CPM 2	12	9			
CPM 2	13	14			
CPM 2	14	21			
CPM 2	15	10			
CPM 2	16	12			
CPM 2	17	7			
CPM 2	18	3			
CPM 3	1	5	6		
CPM 3	2	10			
CPM 3	3	13			
CPM 3	4	14			
CPM 3	5	7	12		
CPM 3	6	8	9	10	
CPM 3	7	11			
CPM 3	8	11			
CPM 3	9	12			
CPM 3	10	12	13	14	
CPM 3	11	17			
CPM 3	12	15			
CPM 3	13	15	16		
CPM 3	14	16	17		
CPM 3	15	17			
CPM 3	16	18			
CPM 3	17	18			
CPM 4		END TEST NETWORK A			

21 in CKTA. Both networks are used in the test decks available with the CPM003 as well as the CPM004 package. Tables 3 and 4 give the outputs to be expected from NETA and CKTA, respectively. Table 5 displays the input required for NETA; that for CKTA would, of course, be similar. Although the node numbering in NETA is in topological order, it should be clear from the preceding discussion that this is unnecessary. Node numbers can be reassigned in any way, the type-2 and type-3 cards can be entered in any sequence, and the program produces the same results as long as duplicate node numbers are avoided, the node numbers are reassigned consistently, and no node number higher than 18 is used—the number specified as maximum on the type-1 card for this job. If larger node numbers are used and the maximum node number is changed to

Table 6 Stages in processing NETA

A (RDCPIO)	B (RDCPIO)	C (PSCT)	C (LGTM)	C (CCP)
1. 0 00005 0 00001	1. 0 00005 4 00006	1. 0 00000 0 00002	1. 2 00005 2 00076	1. 2 00005 2 00002
2. 0 00017 0 00002	2. 4 00012 4 00015	2. 0 00000 0 00001	2. 2 00017 2 00075	2. 2 00017 2 00003
3. 0 00036 1 00002	3. 4 00016 0 00007	3. 0 00000 0 00001	3. 2 00036 2 00100	3. 2 00036 2 00000
4. 0 00024 0 00003	4. 4 00014 0 00010	4. 0 00000 0 00001	4. 2 00024 2 00070	4. 2 00024 2 00010
5. 0 00014 1 00003	5. 0 00011 4 00012	5. 0 00001 0 00002	5. 2 00021 2 00071	5. 2 00021 2 00007
6. 0 00006 1 00004	6. 4 00013 4 00013	6. 0 00001 0 00003	6. 2 00013 2 00064	6. 2 00013 2 00014
7. 0 00030 0 00006	7. 4 00014 0 00014	7. 0 00001 0 00001	7. 2 00051 2 00055	7. 2 00051 2 00023
8. 0 00010 1 00006	8. 0 00015 4 00016	8. 0 00001 0 00001	8. 2 00023 2 00035	8. 2 00023 2 00043
9. 0 00004 0 00007	9. 4 00021 4 00017	9. 0 00001 0 00001	9. 2 00017 2 00041	9. 2 00017 2 00037
10. 0 00012 1 00007	10. 0 00017 4 00020	10. 0 00002 0 00003	10. 2 00031 2 00056	10. 2 00031 2 00022
11. 0 00013 0 00011	11. 0 00020 4 00021	11. 0 00002 0 00001	11. 2 00064 2 00025	11. 2 00064 2 00053
12. 0 00011 1 00011	12. 4 00021 4 00022	12. 0 00003 0 00001	12. 2 00042 2 00035	12. 2 00042 2 00043
13. 0 00016 0 00012	13. 4 00022 0 00000	13. 0 00002 0 00002	13. 2 00054 2 00042	13. 2 00054 2 00036
14. 0 00025 0 00013		14. 0 00002 0 00002	14. 2 00056 2 00044	14. 2 00056 2 00034
15. 0 00012 0 00014		15. 0 00002 0 00001	15. 2 00066 2 00024	15. 2 00066 2 00054
16. 0 00014 1 00014		16. 0 00002 0 00001	16. 2 00072 2 00017	16. 2 00072 2 00061
17. 0 00007 0 00015		17. 0 00003 0 00001	17. 2 00075 2 00012	17. 2 00075 2 00066
18. 0 00003 0 00000		18. 0 00002 0 00000	18. 2 00100 2 00003	18. 2 00100 2 00075

be greater than or equal to the largest node number about which information is entered, the program works properly, but prints out vacuous information about all node numbers that are not actually used in the problem.

Table 6 shows how the A and B arrays are stored for NETA, and what is stored in the C array at the end of the PSCT, LGTM, and CCP subprograms. Note that the digits displayed are octal and that a 4 in a prefix or tag digit represents a flag bit in the leftmost position, a 2 represents a flag bit in the middle position and a 1 represents a flag bit in the rightmost position.

For illustration, consider the entry in the third row of the A column. The octal 36 in the decrement portion of the word represents the decimal number 30, the duration of activity 3 in NETA. (Duration times are shown above the nodes in both the NETA and CKTA diagrams.) The 1 in the tag portion of this word signals that the index identifying the first successor to Node 3 can be found in the righthand portion of the word whose relative position in the B array is given by the number 2 in the address portion of the same word.

Going to the second word of the B array, we find that the address portion contains octal 15 (decimal 13). From either the diagram or input listing, we can verify the fact that node 13 is indeed an immediate successor of node 3.

The 4 in the tag position of the second word in the B array signals that the successor in the address portion is the last of this particular set of successors—in this case, the only one. Again, we can verify this.

Except for overlay, the A and B arrays are constant throughout the critical-path calculation. C, however, is variable. At the end of the PSCT subprogram, a word in the C array specifies the number of predecessors and successors possessed by the node whose number is equal to the relative position of the word in the C array. The number of predecessors is to be found in the decrement portion of the C word; the number of successors in the address portion. For example, the tenth word in the C array has 2 stored in its decrement and 3 stored in its address. This informs us that Node 10 has two predecessors and three successors in NETA.

After LGTM (the main algorithm), the C array contains—in the case of a consistent network—the value 2 in the prefix and tag portions of all its words. (This is equivalent to q and t vectors whose components have all been made 1.) The decrement portions then contain forward times, the address portions backward times.

After CCP, the decrement portions of the A array contain early start times (not shown). The decrement portions of the C array contain early finish times (equivalent to what we have been calling forward times). The address portions of the C array contain late start times. The C array as shown in Table 4 is also copied into corresponding relative positions in the B array.

The overwriting of A and B positions was inspired by the prime motivation of this particular programming effort—to make as much space as possible available for storing information about the network. Minor modifications to the CCP and CTOB routines would eliminate the overwriting and retain the precedence information for such future applications as resource allocation.

From the information stored by CCP in the A and B arrays, the output subprograms CPMOUT/WRCPM produce the report shown in Table 3. Most of the column headings in the report are self-explanatory, except for TFLOAT and FFLOAT. TFLOAT is the difference between early and late start times. In the case of activities on the critical path, this difference is zero; these activities are identified by rows of asterisks on the report. FFLOAT is the difference between an early finish time for an activity and the early start time of that one or more of its successors whose early start time is smallest. Node 4, for example, has an early finish time of 20. Its successor, Node 14, has an early start time of 25. The difference is shown as FFLOAT for Node 4.

In the case of inconsistent networks, subprogram LGTM cannot store the value 2 in all the prefixes and tags of the C array. For example, Table 7 shows how the C array is left at the end of the LGTM program for CKTA.

A prefix of 2 indicates that a forward time has been successfully calculated; a tag of 2 indicates that a backward time has been successfully calculated. In these cases, the corresponding decrements and addresses display times. Prefixes and tags of 0, however, indicate that time calculations could not be made. In these cases, the decrements corresponding to 0 prefixes contain a number that specifies how many predecessor arrows (pointing directly at a given

Table 7 Matrix C of CKTA at end of LGTM processing

			C (LGTM)		
1.	2	00005	0	00002	ENTRY PATH
2.	2	00017	2	00075	
3.	2	00036	2	00100	
4.	2	00024	2	00070	
5.	2	00021	0	00001	ENTRY PATH
6.	0	00001	0	00001	CIRCUIT NODE
7.	2	00051	0	00001	ENTRY PATH
8.	0	00001	0	00001	CIRCUIT NODE
9.	0	00001	2	00041	EXIT PATH
10.	0	00001	2	00056	EXIT PATH
11.	0	00001	0	00001	CIRCUIT NODE
12.	0	00002	2	00035	EXIT PATH
13.	0	00001	2	00042	EXIT PATH
14.	0	00001	2	00044	EXIT PATH
15.	0	00002	2	00024	EXIT PATH
16.	0	00002	2	00017	EXIT PATH
17.	0	00003	2	00012	EXIT PATH
18.	0	00002	2	00003	EXIT PATH
19.	2	00002	2	00026	
20.	2	00005	2	00024	
21.	2	00026	2	00021	

Table 8 Prefix and tag values

Prefix	*Tag*	
0	0	Node in circuit or between two or more circuits
2	0	Node points at circuit
0	2	Node is pointed to by circuit
2	2	Node has no connection with circuit

node) could not be removed by the algorithm; similarly, 0 tags identify address portions that specify how many successor arrows could not be removed by the algorithm. Prefix and tag values are shown in Table 8. The first three cases are identified by the corresponding entries "circuit node," "entry path," and "exit path" in Table 7.

The printout in the case of inconsistent circuits is exemplified by Table 4. Nodes that have neither successors nor predecessors in the circuit are identified by asterisks in the first column. Circuit nodes are identified by pairs of entries in the first column: the first of these gives the number of unremoved predecessors; the second gives the number of unremoved successors. Entry nodes (second column) have pairs of entries of which the first is forward time and the second is the number of unremoved successors. Exit nodes (third column) have pairs of entries of which the first is the number

Table 9 CPM002 calculation times (in seconds) for networks of various sizes

Subroutine	NETA	CKTA	D10	D100
RNC/RNDM1	....	....	3.902	3.946
PSCT	0.016	0.016	1.868	1.909
LGTM	0.034	0.031	35.554	344.622
CCP	0.004	....	2.075	2.107
TOTAL (incl. I/O)	10		85	395

of unremoved predecessors and the second is backward time. The printout for CKTA thus tells us that Nodes 6, 8, and 11 form a circuit; Nodes 1, 5, and 7 point (directly or indirectly) to nodes in a circuit, Nodes 9, 10, 12, 13, 14, 15, 16, 17, and 18 are pointed at (directly or indirectly) by nodes in a circuit, and the remaining nodes have no connection with any circuit.

results

The primary purpose of program CPM002 is high-speed calculation of the critical paths of large networks. However, the resulting program is relatively efficient on small networks if enough of these are batched to make economical use of a large computer. To illustrate calculation speeds for small networks, times for NETA and CKTA calculations performed on a 7090 are shown in Table 9. No CCP time is shown for CKTA since subroutine CCP is not called for an inconsistent network.

To test calculation speeds on large networks, two subroutines were written: RNC (which generates and stores a precedence matrix) and RNDM1 (which generates a duration vector). The characteristics of the matrix and vector are determined by parameter values set by the main program. As a matter of interest, the times required by these programs to generate two large networks, D10 and D100, are also shown in Table 9.

The basic calculation time (exclusive of input/output) required by CPM002 depends primarily on two factors, the size of the basic arrays and the number of iterations of subroutine LGTM required for a solution. The latter depends upon the "diameter" of the network—the largest number of consecutive arrows connecting a source and a sink. Network D10 consists of 3000 nodes and 32,400 arrows; it has a diameter of 10, i.e., 10 iterations are required by the LGTM subroutine. Network D100 consists of 3000 nodes and 32,670 arrows; it has a diameter of 100. Since D100 is virtually of the same size as D10 and requires ten times as many iterations, the LGTM calculation time for D100 is—as would be expected—approximately ten times as long as for D10.

NETA and CKTA were run as a batch; the total time of just under 10 seconds included the time required to read 80 card images and write 300 lines. The total time required for D10 was about a minute and a half, and for D100 about six minutes and a half; in both cases,

this included the writing of about 4000 lines; the ratio of the LGTM calculation times (35 seconds for the first, and 345 seconds for the second) was approximately as expected.

To test the program's efficiency in detecting and providing information about circuits in large networks, a program was written that introduced an inconsistency into network D10. The inconsistency was a circuit containing nodes 400, 709, 710, 711, and 1020. As a consistent network, D10 had a total calculation time of just under 85 seconds; as an inconsistent network, this time was reduced to just under 57 seconds. This included the time to print out node information for all 3000 nodes in the format shown in Table 4. The author has no direct connection with the actual use of large critical-path networks and, as a consequence, no way to determine whether the smaller or larger diameter is more characteristic of a network of the size of D10 or D100.

miscellaneous programming comments

About 1400 words were required to store the programs that make up CPM002. In addition, about 2100 words were needed for the portions of IBSYS that were constantly in residence in the 7090 for which the program was written, and about 7000 words were required for FORTRAN input/output subroutines. This left about 22,000 words for the storage of arrays and buffers. For the calculation of D10 and D100 on the 7090, 22,500 words were reserved for array storage; this was made possible by the overlay feature of the IBSYS loader. Use of the overlay made about 1500 words available for I/O buffers. Even more storage can be made available by several methods, as—for example—specifying ALTIO on the $IBJOB card.

The calculation subroutines (PSCT, LGTM, and CCP) require only 280 words of storage; the longest of them, LGTM, requires half of this. Readers interested in processing much larger networks than those discussed in this paper may find it useful to consider a three-pass operation which would generate the A and B arrays and store them on tape in binary form, read them back in for processing, and then write out the results.

CPM002 requires two words of storage per node. The duration vector d is stored in the left half of the words making up the A array. Vectors r, q, p, a, u, t, s, b have their individual components combined into single words in the C array. The remaining halfword is used for a "pointer" to the first successor of the node which it describes. It is stored in the right half of a word in the A array. An earlier version of this program did not use this pointer and thus required only 1 1/2 7090 words of storage per node. The pointer was introduced to gain greater efficiency in using the subroutines for problems more general than critical-path calculations. For critical-path calculations even larger than those considered here, the storage requirements can be reduced to one word per node by doing forward and backward calculations consecutively rather than concurrently.

CPM002 does not include a check to determine whether input information exceeds the bounds of the B array. However, it would be fairly easy to incorporate such a check.

A minor modification to LGTM, changing the forward and backward time selection criterion to smallest rather than largest is incorporated in a subroutine called STIM. For directed graphs without circuits, use of this subroutine permits calculation of the shortest rather than the longest path through a network.

If all the activity durations are considered to be unity, CPM002 can be used to give information about the structure of any directed graph. Using this idea, a program named IRA (Identify Redundant Arcs) is available which, in conjunction with PSCT and LGTM, flags all redundant arrows (arcs) in the B array.

Concluding remarks

The objective in writing CPM002 was to investigate ideas rather than to develop programs (which are, in a sense, incidental). The utility of network programs intended for actual use is measured primarily by how effectively they provide information to management and how well they can be used as instruments of management control. Readers interested in these aspects of critical-path programs are referred to the publications of the Construction Institute, Stanford University.[6,7]

To the best of the author's knowledge, the time required by CPM002 to calculate the critical paths of networks D10 and D100 is shorter than that of many programs that just do topological sorting of networks of this size. Kahn,[8] for example, gives an estimated time of 40 to 50 minutes on the 7090 for topological sorting of a network of 30,000 arrows. In contrast, it takes CPM002 only 1 1/2 minutes for D10—and 6 1/2 minutes for D100—to do the entire calculation, and 57 seconds to locate the loop and to print out diagnostic information for the inconsistent D10 case. However, it is not clear that a direct comparison of this kind is valid, since Kahn's algorithm is concerned with event node networks whereas CPM002 works on activity node networks. The size comparison is valid, but the activity numbering conventions in the event node representation are such that, since two nodes are required to represent an activity, the CPM002 scheme of using node numbers as relative addresses is not directly applicable. Actual use of CPM002 would, in any event, require input/output routines to provide alphabetical information and displays of the kind provided by SPRED-CPM, the program described in Baker.[7] The author believes that conversion of an event node representation to an activity node representation could be programmed in such a way as to add an insignificant amount of time to these routines; but no programming has been done to support this conjecture.

ACKNOWLEDGMENT

The program described in this paper is a specific outgrowth of research on the general topic of "problem definition methods" started in collaboration with Professor Teichroew of the Graduate

School of Business, Stanford University (now at Case Institute of Technology), in 1963. At that time, Mr. J. P. Seagle, a GSB graduate student, programmed a FORTRAN II version of the basic algorithm described in this paper for purposes of comparison with a MAP (Macro-Assembly Program) version, called CPM001, programmed by me. This and other early work has been reported in oral presentations at various times, but the present write-up of a derivative program (CPM002) is the first written report to describe any part of our work together.

Though the program and this paper are the results of work done on my own, I would like to acknowledge my indebtedness to Professor Teichroew and Mr. Seagle for the stimulating partnership which they provided in the early stages; it contributed substantially to this paper.

The early stages of my research work at Stanford University were made possible by a grant of the University's Computation Center. I should like to express my gratitude both for the grant itself and for the cooperation and helpfulness of the Computation Center staff.

CITED REFERENCES AND FOOTNOTES

1. B. Dimsdale, "Computer construction of minimal project networks," *IBM Systems Journal* **2**, 24–36 (March 1963).
2. K. E. Iverson, *A Programming Language*, John Wiley (1962).
3. K. E. Iverson, "Formalism in programming languages," *Communications of the Association for Computing Machinery* **7**, 2, 80–88 (February 1964).
4. It is hoped that this brief discussion conveys some idea of what an efficient notation can contribute to formalizing the design of efficient programs. Since the objective of this paper is primarily to describe a specific program, those interested in the question of efficient equipment and programming design are referred to References 2 and 3. The Appendix gives a step-by-step display of the operation of Steps 1 through 6 of the Iverson algorithm for the three iterations required to produce a new-origin vector with no non-zero components
5. This figure has been copied with minor modifications from John W. Fondahl, *A Non-Computer Approach to the Critical Path Method for the Construction Industry*, Technical Report No. 9, The Construction Institute, Stanford University.
6. The consistent network NETA was taken directly from J. W. Fondahl, Ibid.
7. C. W. Baker, SPRED-CPM, *A Computer Program for the Solution of the Precedence Diagram using Critical Path Methods*, Technical Report No. 56, The Construction Institute, Stanford University, Stanford, California. This report describes an extremely efficient and flexible program which provides excellent project timing information in the form of labelled bar charts, resource schedule information in both graphical and digital form, flexible subreports, and a variety of other useful features. The computer program itself is currently available only to members of the Construction Institute, but the description of the program in the Baker report is, in itself, informative and useful.
8. A. B. Kahn, "Topological sorting of large networks," *Communications of the Association for Computing Machinery* **5**, 11, 558–562 (November 1962).

Appendix — A step-by-step analysis of the Iverson-language algorithm as it applies to NETA

This Appendix is presented for readers interested in illustrations of the Iverson-language operators. The concentration is on the first six steps of the Iverson-language program shown in Figure 6. At the outset of the program, we have, in addition to the precedence matrix P and the duration vector d, the predecessor vector p (calculated by summing the columns of P) and the successor vector s (calculated by summing the rows of P). The first six steps are concerned only with p; the next six steps process s in a directly analogous manner.

The convention in interpreting Iverson-language statements is to scan them from right to left. Thus, in the first statement, $p = 0$ has the effect of the relational statement $(p = 0)$ described in Iverson.[2] In other words, it defines a logical vector whose components have the value 1 when the corresponding values of p are zero, and the value 0 when the corresponding values of p are not zero.

1. $r \leftarrow \bar{q} \wedge p = 0$
   $r \leftarrow (1,1,1,1,1,1) \wedge (2,0,1,1,2,0) = (0,0,0,0,0,0)$
   $r \leftarrow (1,1,1,1,1,1) \wedge (0,1,0,0,0,1)$
   $r \leftarrow (0,1,0,0,0,1)$

2. $0: \vee/r = \rightarrow$ Branch to Step 7
   $0: \vee/(0,1,0,0,0,1)$
   $0: 1$ ($\neq$, No Branch)

3. $a \leftarrow /a;r;d/$
   $a \leftarrow /(0,0,0,0,0,0);(0,1,0,0,0,1);(15,5,30,20,10,7)/$
   $a \leftarrow (0,5,0,0,0,7)$

4. $a \leftarrow a + r \times a \overset{r}{\times} P$
   $a \leftarrow (0,5,0,0,0,7) + (0,1,0,0,0,1) \times (0,5,0,0,0,7) \overset{r}{\times} \begin{pmatrix} 000000 \\ 000110 \\ 000000 \\ 100000 \\ 101000 \\ 000010 \end{pmatrix}$
   $a \leftarrow (0,5,0,0,0,7) + (0,1,0,0,0,1) \times (0,0,0,5,7,0)$
   $a \leftarrow (0,5,0,0,0,7) + (0,0,0,0,0,0)$
   $a \leftarrow (0,5,0,0,0,7)$

5. $q \leftarrow r \vee q$
   $q \leftarrow (0,1,0,0,0,1) \vee (0,0,0,0,0,0)$
   $q \leftarrow (0,1,0,0,0,1)$

6. $p \leftarrow p - r \overset{+}{\times} P$

$p \leftarrow (2,0,1,1,2,0) - (0,1,0,0,0,1) \overset{+}{\times} \begin{pmatrix} 000000 \\ 000110 \\ 000000 \\ 100000 \\ 101000 \\ 000010 \end{pmatrix}$

$p \leftarrow (2,0,1,1,2,0) - (0,0,0,1,2,0)$

$p \leftarrow (2,0,1,0,0,0)$

(END OF ITERATION 1)

1. $r \leftarrow \bar{q} \wedge p = 0$

$r \leftarrow (1,0,1,1,1,0) \wedge (2,0,1,0,0,0) = (0,0,0,0,0,0)$

$r \leftarrow (1,0,1,1,1,0) \wedge (0,1,0,1,1,1)$

$r \leftarrow (0,0,0,1,1,0)$

2. $0: \wedge/r = \rightarrow 7$

$0: \vee/(0,0,0,1,1,0)$

$0: 1\ (\neq)$ No Branch

3. $a \leftarrow /a; r; d/$

$a \leftarrow /(0,5,0,0,0,7); (0,0,0,1,1,0); (15,5,30,20,10,7)/$

$a \leftarrow (0,5,0,20,10,7)$

4. $a \leftarrow a + r \times a \overset{r}{\times} P$

$a \leftarrow (0,5,0,20,10,7)$

$+ (0,0,0,1,1,0) \times (0,5,0,20,10,7) \overset{r}{\times} \begin{pmatrix} 000000 \\ 000110 \\ 000000 \\ 100000 \\ 101000 \\ 000010 \end{pmatrix}$

$a \leftarrow (0,5,0,20,10,7) + (0,0,0,1,1,0) \times (20,0,10,5,7,0)$

$a \leftarrow (0,5,0,20,10,7) + (0,0,0,5,7,0)$

$a \leftarrow (0,5,0,25,17,7)$

5. $q \leftarrow r \vee q$

$q \leftarrow (0,0,0,1,1,0) \vee (0,1,0,0,0,1)$

$q \leftarrow (0,1,0,1,1,1)$

6. $p \leftarrow p - r \overset{+}{\times} P$

$p \leftarrow (2,0,1,0,0,0) - (0,0,0,1,1,0) \overset{+}{\times} \begin{pmatrix} 000000 \\ 000110 \\ 000000 \\ 100000 \\ 101000 \\ 000010 \end{pmatrix}$

$p \leftarrow (2,0,1,0,0,0) - (2,0,1,0,0,0)$

$p \leftarrow (0,0,0,0,0,0)$

(END OF ITERATION 2)

1. $r \leftarrow \bar{q} \wedge p = 0$
   $r \leftarrow (1,0,1,0,0,0) \wedge (1,1,1,1,1,1)$
   $r \leftarrow (1,0,1,0,0,0)$

2. 0: $\vee/r = \rightarrow 7$
   0: $\vee/(1,0,1,0,0,0)$
   0: 1($\neq$) No Branch

3. $a \leftarrow /a; r; d/$
   $a \leftarrow /(0,5,0,25,17,7); (1,0,1,0,0,0); (15,5,30,20,10,7)/$
   $a \leftarrow (15,5,30,25,17,7)$

4. $a \leftarrow a + r \times a \overset{r}{\times} P$
   $a \leftarrow (15,5,30,25,17,7) + (1,0,1,0,0,0) \times (15,5,30,25,17,7) \overset{r}{\times} \begin{pmatrix} 000000 \\ 000110 \\ 000000 \\ 100000 \\ 101000 \\ 000010 \end{pmatrix}$
   $a \leftarrow (15,5,30,25,17,7) + (1,0,1,0,0,0) \times (25,0,17,5,7,0)$
   $a \leftarrow (15,5,30,25,17,7) + (25,0,17,0,0,0)$
   $a \leftarrow (40,5,47,25,17,7)$

5. $q \leftarrow r \vee q$
   $q \leftarrow (1,0,1,0,0,0) \vee (0,1,0,1,1,1)$
   $q \leftarrow (1,1,1,1,1,1)$

6. $p \leftarrow p - r \overset{+}{\times} P$
   $p \leftarrow (0,0,0,0,0,0) - (1,0,1,0,0,0) \overset{+}{\times} \begin{pmatrix} 000000 \\ 000110 \\ 000000 \\ 100000 \\ 101000 \\ 000010 \end{pmatrix}$
   $p \leftarrow (0,0,0,0,0,0) - (0,0,0,0,0,0)$
   $p \leftarrow (0,0,0,0,0,0)$
   (END OF ITERATION 3)

1. $r \leftarrow \bar{q} \wedge p = 0$
   $r \leftarrow (0,0,0,0,0,0) \wedge (1,1,1,1,1,1)$
   $r \leftarrow (0,0,0,0,0,0)$

2. 0: $\vee/r = \rightarrow 7$
   0: $\vee/(0,0,0,0,0,0)$
   0: 0 (=) Branch to 7. Forward calculation complete.

Note that Step 4 in each of the iterations leaves forward times in the a vector. These forward times are those associated with the nodes in the successive iterations pictured in Figure 3.