# Methods of computing event times in project networks

*By* D. Nudds*

**Digital computer methods for PERT event time calculations may be based on the familiar methods used for hand computations, in which case the network must be topologically ordered. For the calculation of the earliest and latest event times it is sufficient to order the arcs in one of several ways. An ordering may be achieved by using the topological structure in a way analogous to the hand computation method. Alternatively, iterative methods may be used; these compare favourably provided the network data is presented in a suitable form.**

(First received May 1968 and in revised form July 1968)

The basic techniques of project scheduling by using networks are well known under the names of PERT, Critical Path Analysis, etc.—see, for example, Robinson (1963), Kelly (1961), Lockyer (1967). The computation of event times is a constituent part of all project scheduling programs and of their developments into more general techniques for resource allocation (RAMPS), cost-time networks (PERT/COST), etc. Most accounts of the methods of computing event-times are descriptions of the simple hand-computational method. Kahn (1962) describes one method for sorting the activities in a network and Parikh and Jewell (1965) describe a method for decomposing larger networks into several sub-networks. Klein (1967) describes an implementation of Kahn's method within a RAMPS program.

In this paper we describe and compare the basic computational techniques. The problem, in graph-theoretic terms may be expressed as follows, using the terminology as defined by Busacher and Saaty (1965). A project is represented by a network consisting of vertices, representing events, connected by arcs, representing activities.

We are given a 'network' which we define as a single component, simple, directed graph with no cycles. A vertex which has no arcs incident to it we call an *origin*. A vertex with no arcs incident from it, we call a *terminus*. There must obviously be at least one origin and at least one terminus. Each arc has a non-negative length, and the sum of the arc lengths along a path is the *path-length*. Considering all the paths between any origin and any terminus, a path with maximum length is a *critical path*. Projects with $m$ origins and $n$ termini thus have at least $m \times n$ critical paths.

## Formulation of the problem

Although some applications use networks with several origins and several termini, we have restricted our analysis to the situation with one origin and one or more termini. Many applications either fall directly into this category or may be made to do so by extending the network by the inclusion of a new origin and several extra arcs, each leading from the new origin to one of the old origins. Failing this, cases in which more than one origin are present, if they should arise, may be dealt with by an extension of the single-origin method. We can define the function $E(v)$ of any vertex $v$ as the maximum of all the path-lengths from the origin to $v$: this corresponds to the earliest event time in scheduling terminology. Similarly, corresponding to latest event time, we define $F(v)$ for any vertex $v$ as the minimum of the difference between $E(t)$ and the path-length from $v$ to $t$, for all paths, if any, from $v$ to $t$, and for all termini $t$.

We use the notation:

$V$ denotes the set of all vertices of the network;
$T$ denotes the set of all termini;
$\phi$ denotes the origin;
$A$ denotes the set of arcs;
$\lambda(a)$ denotes the length of an arc $a$;
$\lambda(p)$ denotes the length of a path $p$;
$\mathscr{I}n(v)$ denotes those arcs incident to vertex $v$;
$\mathscr{E}x(v)$ denotes those arcs incident from vertex $v$;
$I(a)$ denotes the starting vertex of arc $a$;
$J(a)$ denotes the terminal vertex of arc $a$;
$P(v, w)$ denotes the set of all paths between vertices $v$ and $w$.

Hence we have, as previously defined

$$\left.\begin{aligned} E(v) &= \operatorname*{Max}_{p \in P(\phi, v)} \{\lambda(p)\} \qquad \text{where } E(\phi) = 0 \\ \text{and} \quad F(v) &= \operatorname*{Min}_{\substack{p \in P(v, t) \\ t \in T}} \{E(t) - \lambda(p)\} \end{aligned}\right\} (1)$$

The basis of all computations is

$$\left.\begin{aligned} E(v) &= \operatorname*{Max}_{a \in \mathscr{I}n(v)} \{E(I(a)) + \lambda(a)\} \\ \text{and} \quad F(v) &= \operatorname*{Min}_{a \in \mathscr{E}x(v)} \{F(J(a)) - \lambda(a)\} \\ \text{with} \quad E(\phi) = 0 &\text{ and } E(t) = F(t) \text{ for all } t \in T. \end{aligned}\right\} (2)$$

A hand-calculation method using a diagrammatic representation of the network, is based on these equations. It is simple and straightforward because, starting with $E(\phi) = 0$, it is easy at any stage to select a vertex such that $E(I(a))$ is known for all $a \in \mathscr{I}n(v)$; for the calculation of the $F$ values, starting from termini it is again at any stage easy to select a vertex for which $F(J(a))$ is known for all $a \in \mathscr{E}x(v)$. For networks of up to a hundred or so activities, this is not onerous—the visual scanning involved is easy and fast provided the diagram has been

* *Computing Laboratory, University of Bradford, Bradford, Yorks.*

properly drawn: the only real labour is in performing the many very simple arithmetic operations.

## Automatic computation: the representation of networks

Examining now the implementation of critical path scheduling on a digital computer, we see immediately that there are several alternative ways in which the network information may be stored. In any case, each vertex $v_i$ may be identified by the integer index $i$, and each arc $a_j$ by its index $j$: we assume that on the whole the values of the indices range from 1 upwards, with few gaps. As far as the vertices are concerned, this can simply be a labelling requirement imposed on the data, or the result of an initial transformation.

In the following we now write $I_r$, $J_r$, $E_r$, $F_r$ and $\lambda_r$ for $I(a_r)$, $J(a_r)$, $E(v_r)$, $F(v_r)$ and $\lambda(a_r)$ respectively. Three representations are here suggested:

R1: An arc array. In effect 3 vectors $I$, $J$, $\lambda$ with components $I_r$, $J_r$, and $\lambda_r$.

R2: Incidence matrix, or $\lambda$ - array. Here is stored a square array of elements $d_{ij} = \lambda(a)$ if there is an arc $a = (v_i, v_j)$. and otherwise a special symbol, (e.g. $-1$), indicating absence of an arc. In any real network there is, of course, a preponderance of null entries, and a large network, apart from being exceedingly sparse, would require an inordinate amount of storage space, certainly too much for most immediate access stores.

R3: A more complex structure in which the null entries are removed from R2: any one of the usual ways of representing a sparse array may be employed, e.g.

R3($a$): For the method we shall describe later it is suitable to group together the non-null entries according to their rows.

R3($b$): For other methods a grouping according to columns may be better. Klein (1967) implements each of these groups in the form of a list structure. Further groupings may be useful in other methods; in effect R3($a$) gives access to the immediate successors of any vertex, and R3($b$) to the predecessors.

Now consider the method implied by equations (2) above. At any stage in the calculation of the components of $E$, several components are known, corresponding to vertices in the set $V_k$, say, with $V_k \subset V$.

Then for a further value to be found it is necessary to find a vertex $v_{k+1}$ such that $\{I(a); a \in \mathscr{I}n\,(v_{k+1})\} \subset V_k$ and then $V_{k+1} = \{v_{k+1}\} \cup V_k$; i.e. the set is extended by the further inclusion of a vertex the immediate predecessors of which are already in the set. Obviously in this case representations R1 and R2 can involve many scans of the whole of the stored information. Representation R3($b$), however, by giving ready access to the members of the set $\{I(a); a \in \mathscr{I}n\,(v_r)\}$, for each $v_r$, leads to a more efficient algorithm. Several searches through the various lists may still be necessary and we shall later describe yet more efficient methods: these methods involve ordering the given information in some suitable way. However, an alternative approach will now be described.

## Iterative methods

In this method the most suitable form of storage of the data is R1, an arc array, the $r$th entries corresponding to arc $a_r$ being $I_r$, $J_r$ and $\lambda_r$. To calculate the values $E_r$ the algorithm starts by initially setting $e_r = 0$, for all $r$.

Then for each arc in turn, if $e_{I_r} < e_{J_r} + \lambda_r$ set $e_{J_r} = e_{I_r} + \lambda_r$. Repeat the process until no further changes in any values are recorded. The iteration obviously converges since, at any stage, the value of $e_i$ represents, if, non zero, the length of a path terminating at $v_i$. Provided there are no cycles in the graph then the values of $e_i$ tend to a limit which is the length of the longest path terminating at $v_i$, i.e., to $E_i$.

The calculation of the values of $F_i$ proceeds similarly. First set $f_t = E_t$ for all terminals $v_t$ then for each arc $a_r$ in turn set $f_{I_r}$ to the value $f_{J_r} - \lambda_r$ if this is smaller, again iterating until convergence results.

The speed of convergence of the calculation is obviously highly dependent on the way in which the arc data is ordered. The closer the data is to an optimum topological ordering, which we define later, the smaller is the number of iterations that are required.

If the information is presented in a more or less random order it might be conjectured that since there is then no particular preferential direction in which to scan the arc data, it may be preferable to perform the iteration with a sequence of scans in alternating directions. Trials of this method, as described later, have shown the conjecture to be false.

On the other hand, realistic data is not normally randomly ordered. Without any difficulty, and probably as a matter of course, a project analyst will present his data in such a way that if there is a path in which arc $a_i$ occurs earlier than arc $a_j$ then it is more likely that $i < j$. Even when this is only a mild tendency, it is sufficient to guarantee that an iteration consisting of arc-scans in the same direction is the faster.

## Ordering of arcs

Further consideration of the convergence of the iterative method shows that in fact one scan of the arcs is sufficient to determine all $E_i$, and a further scan in the opposite direction determines all $F_i$, provided the arcs are suitably ordered in the first place. In this case the method is more or less the same as the non-iterative method first described: in each case we require a topological ordering of the network, in which for two arcs $a_i$ and $a_j$, if $i > j$ then there cannot exist any path from $J(a_i)$ to $I(a_j)$.

An equivalent condition to this is that for any vertex $v$, all members of the set $\mathscr{I}n(v)$ must appear earlier, in the ordered set $A$ of arcs, than members of $\mathscr{E}x(v)$. The converse condition holds, of course, for the calculation of the values of $F$: i.e. in the second set of arcs, for any $v$, all members of $\mathscr{E}x(v)$ must appear before any member of $\mathscr{I}n(v)$. Thus the reverse order to that for the calculation of $E$ suffices.

In the simplest programs written for critical path analysis this order is achieved by restricting the labelling of vertices to integer indices such that if $(v_i, v_j)$ is an arc then $i < j$. It is then sufficient to sort the arcs in order of ascending terminal vertex index. This is directly equivalent to the scanning of the equivalent $\lambda$-array by columns from left to right, the array itself being upper triangular—null on and below the main diagonal. A similar row scan would suffice for the determination of the values of $F$. But these orderings are not necessary: the row-ordering reversed is an example of a possible ordering for the calculation of $F$.

If the restriction is not imposed upon the project analyst that indices should be allocated to vertices in the way described—which is certainly a non-trivial and usually non-acceptable restriction in large networks in practice—then a topological ordering must be achieved by computer.

## Arc ordering by Kahn's method

In each stage of the calculation of the values of $E$, a search is made for a next vertex such that the calculation of $E$ has already been performed for all predecessors of this vertex. Elimination of the scanning of all arcs is achieved by storing the network as a compressed array-row of the form R3($a$). For each vertex a count is initially made of the number of predecessors of the vertex. When a vertex is processed these counts, for all its successors, are diminished by 1. Once the count of any vertex becomes zero it is added to a list which contains the vertices which may be processed next. For the calculation of values of $F$, Kahn described a reversal of this procedure. However, as we have seen this is not necessary; it is sufficient, and simpler, to run through the arcs in the order which is the reverse of the one previously derived.

## Ordering by stage distance

In this method a transformation is effected of the vertex indices so that they are topologically ordered. This requires defining an integer function $f$ over the set of vertices $V$ such that if $v, w \epsilon V$ and there is a path from $v$ to $w$ then $f(v) < f(w)$. Once this has been determined, arranging arcs in order of ascending $f(I(a))$ is a satisfactory arc order. An alternative key for the sort would, of course, be $f(J(a))$.

The simplest function which has the property is the stage distance of a vertex $v$ from the origin, i.e. the maximum number of vertices on any path from the origin to $v$. This is, of course, identical with the definition of $E(v)$ for the case in which all arc lengths are unity and so the stage distance must be found by one of the previously discussed methods: it would be pointless to use Kahn's arc ordering method as an intermediate stage in producing an arc order; on the other hand the iterative method is immediately attractive as a single iteration can be used to produce a vertex order, by stage distance, and hence an arc order, which can be used for both calculation of $E$ and of $F$.

This itself is sufficient, for a large network, to justify the extra time involved in arranging this ordering, in any single calculation. There are other possible benefits: if the arc ordering is itself output for future reinput, then in any future processing of the network with alterations to the arc lengths, the arc order may be used again and the processing time becomes further reduced.

## Calculations and conclusions

A series of programs have been written for the University of Bradford's I.C.T. 1909 Computer. Comparative times for various sets of data are shown in **Table 1**. This shows results for four sets of data.

**Table 1**

**Times in seconds for an arc ordering of 207 arcs**

| METHOD | DATA | | | |
|---|---|---|---|---|
| | ($a$) | ($b$) | ($c$) | ($d$) |
| Kahn's Method | 0·40 | 0·40 | 0·40 | 0·40 |
| Iterations all in same direction | 0·34 | 0·34 | 0·42 | 0·70 |
| Iterations in alternating directions | 0·39 | 0·41 | 0·53 | 1·13 |

($a$) Data containing 207 activities as it was originally set up before the project was started. There was no deliberate attempt to order the data, but in fact the list tended in general to run along paths in the network.

($b$) Data concerning the project some time after it had been started. A number of amendments had been made, distorting somewhat the original order.

($c$) A completely random permutation of the data in ($a$).

($d$) A deliberate series of changes were made to the data ($a$). In effect the list was cut into eight pieces, which were then joined together again in a new order. There was still, of course, a tendency for parts of the original paths in ($a$) to be preserved.

Methods used to sort the arcs were:

(1) Kahn's method of arc ordering.
(2) Iterations, all in the same direction to produce stage distances, and then arc orders.
(3) Iterations in alternating directions to produce stage distances and then arc orders.

It is interesting to note that the deliberate but sectional disordering in data ($d$) gives rise to longer computing times than the completely random order of data ($c$). For the first two sets ($a$) and ($b$), which correspond to realistic computational situations, the simple iterative approach is significantly faster. The actual computing time saved for networks of this size is negligible, compared with overall processing time including input and output and editing. But for larger networks the iterative method is not only faster but more economical in the use of immediate access storage space.

## References

BUSACHER and SAATY (1965). *Finite Graphs with Applications*, McGraw-Hill.
KAHN, A. B. (1962). Topological sorting of Large Networks, *Comm. ACM.*, Vol. 5, pp. 558–562.
KELLY, J. E. (1961). Critical Path Planning and Scheduling: mathematical basis, *Operations Res.*, Vol. 9, pp. 296–320.
KLEIN, M. M. (1967). Scheduling Project Networks, *Comm. ACM.*, Vol. 10, pp. 225–231.
LOCKYER, K. G. (1967). *An Introduction to Critical Path Analysis*, Pitman.
PARIKH, S. C., and JEWELL, W. S. (1965). Decomposition of Project Networks. *Management Sci.*, Vol. 11, pp. 444–459.
ROBINSON, F. D. (1963). The background of the PERT algorithm, *Computer Journal*, Vol. 5, pp. 297–300.