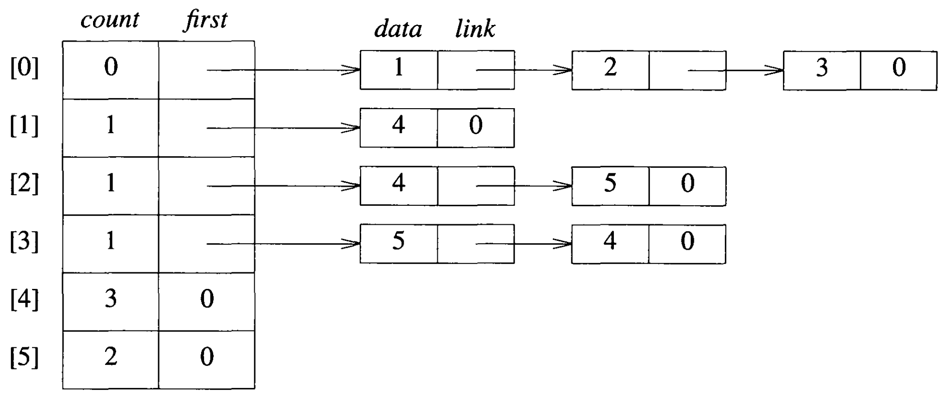


Figure 6.39: Internal representation used by topological sorting algorithm

count has become zero.

Analysis of *topSort*: As a result of a judicious choice of data structures, *topSort* is very efficient. The first **for** loop takes $O(n)$ time, on a network with n vertices and e edges. The second **for** loop is iterated n times. The **if** clause is executed in constant time; the **for** loop within the **else** clause takes time $O(d_i)$, where d_i is the out-degree of vertex i. Since this loop is encountered once for each vertex that is printed, the total time for this part of the algorithm is:

$$O\left(\left(\sum_{i=0}^{n-1} d_i\right) + n\right) = O(e+n)$$

Thus, the asymptotic computing time of the algorithm is $O(e + n)$. It is linear in the size of the problem! □

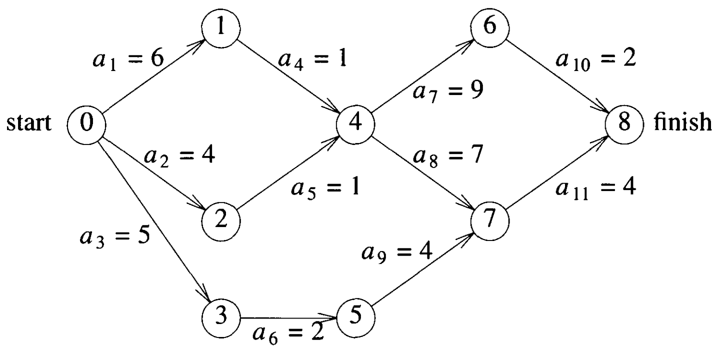
6.5.2 Activity-on-Edge (AOE) Networks

An activity network closely related to the AOV network is the *activity-on-edge*, or *AOE, network*. The tasks to be performed on a project are represented by directed edges. Vertices in the network represent events. Events signal the completion of certain activities. Activities represented by edges leaving a vertex cannot be started until the event at that vertex has occurred. An event occurs only when all activities entering it have been

```
void topSort(hdnodes graph[], int n)
{
  int i,j,k,top;
  nodePointer ptr;
  /* create a stack of vertices with no predecessors */
  top = -1;
  for (i = 0; i < n; i++)
    if (!graph[i].count) {
      graph[i].count = top;
      top = i;
  }
  for (i = 0; i < n; i++)
    if (top == -1) {
      fprintf(stderr,
         "\nNetwork has a cycle. Sort terminated. \n");
      exit(EXIT_FAILURE);
    }
    else {
      j = top;   /* unstack a vertex */
      top = graph[top].count;
      printf("v%d, ",j);
      for (ptr = graph[j].link; ptr; ptr = ptr→link) {
      /* decrease the count of the successor vertices
         of j */
        k = ptr→vertex;
        graph[k].count--;
        if (!graph[k].count) {
        /* add vertex k to the stack */
          graph[k].count = top;
          top = k;
        }
      }
    }
}
```

Program 6.14: Topological sort

completed. Figure 6.40(a) is an AOE network for a hypothetical project with 11 tasks or activities: $a_1, \cdots, a_{11}$. There are nine events: 0, 1, $\cdots$, 8. The events 0 and 8 may be interpreted as "start project" and "finish project," respectively. Figure 6.40(b) gives interpretations for some of the nine events. The number associated with each activity is the time needed to perform that activity. Thus, activity a_1 requires 6 days, whereas a_{11} requires 4 days. Usually, these times are only estimates. Activities a_1, a_2, and a_3 may be carried out concurrently after the start of the project. Activities a_4, a_5, and a_6 cannot be started until events 1, 2, and 3, respectively, occur. Activities a_7 and a_8 can be carried out concurrently after the occurrence of event 4 (i.e., after a_4 and a_5 have been completed). If additional ordering constraints are to be put on the activities, dummy activities whose time is zero may be introduced. Thus, if we desire that activities a_7 and a_8 not start until both events 4 and 5 have occurred, a dummy activity a_{12} represented by an edge <5,4> may be introduced.



(a) Activity network of a hypothetical project

event	*interpretation*
0	start of project
1	completion of activity a_1
4	completion of activities a_4 and a_5
7	completion of activities a_8 and a_9
8	completion of project

(b) Interpretation of some of the events in the network of (a)

Figure 6.40: An AOE network

Activity networks of the AOE type have proved very useful in the performance evaluation of several types of projects. This evaluation includes determining such facts about the project as what is the least amount of time in which the project may be completed (assuming there are no cycles in the network), which activities should be speeded to reduce project length, and so on.

Since the activities in an AOE network can be carried out in parallel, the minimum time to complete the project is the length of the longest path from the start vertex to the finish vertex (the length of a path is the sum of the times of activities on this path). A path of longest length is a *critical path*. The path 0, 1, 4, 6, 8 is a critical path in the network of Figure 6.40(a). The length of this critical path is 18. A network may have more than one critical path (the path 0, 1, 4, 7, 8 is also critical).

The *earliest time* that an event i can occur is the length of the longest path from the start vertex 0 to the vertex i. The earliest time that event v_4 can occur is 7. The earliest time an event can occur determines the *earliest start time* for all activities represented by edges leaving that vertex. Denote this time by $e(i)$ for activity a_i. For example, $e(7)=e(8)=7$.

For every activity a_i, we may also define the *latest time*, $l(i)$, that an activity may start without increasing the project duration (i.e., length of the longest path from start to finish). In Figure 6.40(a) we have $e(6)=5$ and $l(6)=8$, $e(8)=7$ and $l(8)=7$.

All activities for which $e(i)=l(i)$ are called *critical activities*. The difference $l(i)-e(i)$ is a measure of the criticality of an activity. It gives the time by which an activity may be delayed or slowed without increasing the total time needed to finish the project. If activity a_6 is slowed down to take 2 extra days, this will not affect the project finish time. Clearly, all activities on a critical path are strategic, and speeding up noncritical activities will not reduce the project duration.

The purpose of critical-path analysis is to identify critical activities so that resources may be concentrated on these activities in an attempt to reduce project finish time. Speeding a critical activity will not result in a reduced project length unless that activity is on all critical paths. In Figure 6.40(a) the activity a_{11} is critical, but speeding it up so that it takes only 3 days instead of 4 does not reduce the finish time to 17 days. This is so because there is another critical path (0, 1, 4, 6, 8) that does not contain this activity. The activities a_1 and a_4 are on all critical paths. Speeding a_1 by 2 days reduces the critical path length to 16 days. Critical-path methods have proved very valuable in evaluating project performance and identifying bottlenecks.

Critical-path analysis can also be carried out with AOV networks. The length of a path would now be the sum of the activity times of the vertices on that path. By analogy, for each activity or vertex we could define the quantities $e(i)$ and $l(i)$. Since the activity times are only estimates, it is necessary to reevaluate the project during several stages of its completion as more accurate estimates of activity times become available. These changes in activity times could make previously noncritical activities critical, and vice versa.

Before ending our discussion on activity networks, let us design an algorithm to calculate $e(i)$ and $l(i)$ for all activities in an AOE network. Once these quantities are known, then the critical activities may easily be identified. Deleting all noncritical activities from the AOE network, all critical paths may be found by just generating all paths from the start-to-finish vertex (all such paths will include only critical activities and so must be critical, and since no noncritical activity can be on a critical path, the network with noncritical activities removed contains all critical paths present in the original network).

6.5.2.1 Calculation of Early Activity Times

When computing the early and late activity times, it is easiest first to obtain the earliest event time, *ee* [j], and latest event time, *le* [j], for all events, j, in the network. Thus if activity a_i is represented by edge $<k,l>$, we can compute $e(i)$ and $l(i)$ from the following formulas:

$$e(i)=ee[k]$$

and (6.1)

$$l(i)=le[l]\text{–duration of activity } a_i$$

The times *ee* [j] and *le* [j] are computed in two stages: a forward stage and a backward stage. During the forward stage we start with *ee* [0]=0 and compute the remaining early start times, using the formula

$$ee[j]=\max_{i\in P(j)}\{ee[i] + \text{duration of} <i,j>\} \tag{6.2}$$

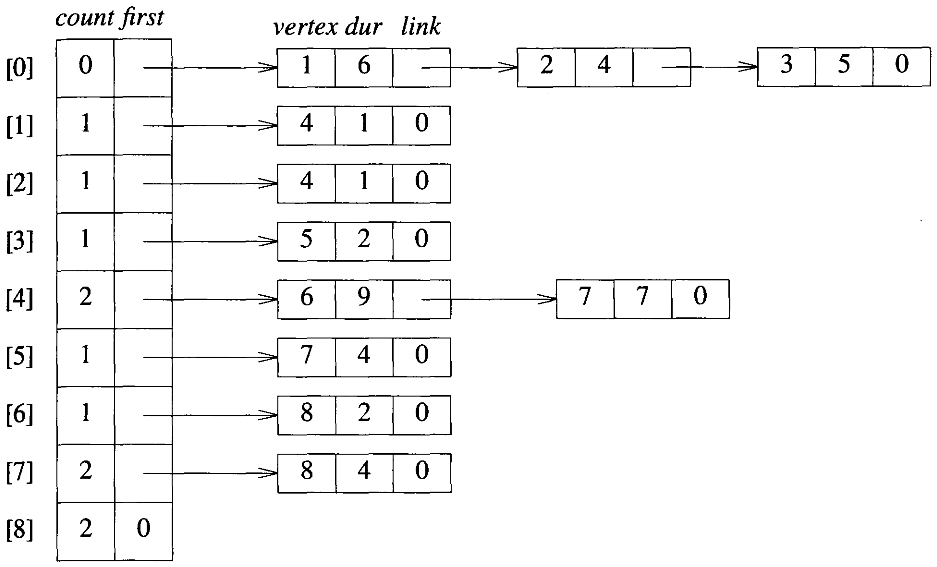
where $P(j)$ is the set of all vertices adjacent to vertex j. If this computation is carried out in topological order, the early start times of all predecessors of j would have been computed prior to the computation of *ee* [j]. So, if we modify *topSort* (Program 6.14) so that it returns the vertices in topological order (rather than outputs them in this order), then we may use this topological order and Eq. 6.2 to compute the early event times. To use Eq. 6.2, however, we must have easy access to the vertex set $P(j)$. Since the adjacency list representation does not provide easy access to $P(j)$, we make a more major modification to Program 6.14. We begin with the *ee* array initialized to zero and insert the code

```
        if (earliest[k] < earliest[j] + ptr→duration)
            earliest[k] = earliest[j] + ptr→duration;
```

just after the line

```
k = ptr→vertex;
```

This modification results in the evaluation of Eq. (6.2) in parallel with the generation of a topological order. *ee* (*k*) is updated each time the *ee* () of one of its predecessors is known (i.e., when *j* is ready for output).



(a) Adjacency lists for Figure 6.40(a)

ee	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
initial	0	0	0	0	0	0	0	0	0	[0]
output 0	0	6	4	5	0	0	0	0	0	[3, 2, 1]
output 3	0	6	4	5	0	7	0	0	0	[5, 2, 1]
output 5	0	6	4	5	0	7	0	11	0	[2, 1]
output 2	0	6	4	5	5	7	0	11	0	[1]
output 1	0	6	4	5	7	7	0	11	0	[4]
output 4	0	6	4	5	7	7	16	14	0	[7, 6]
output 7	0	6	4	5	7	7	16	14	18	[6]
output 6	0	6	4	5	7	7	16	14	18	[8]
output 8										

(b) Computation of *ee*

Figure 6.41: Computing *ee* using modified *topSort* (Program 6.14)

To illustrate the working of the modified *topSort* algorithm, let us try it out on the network of Figure 6.40(a). The adjacency lists for the network are shown in Figure 6.41(a). The order of nodes on these lists determines the order in which vertices will be considered by the algorithm. At the outset, the early start time for all vertices is 0, and the start vertex is the only one in the stack. When the adjacency list for this vertex is processed, the early start time of all vertices adjacent from 0 is updated. Since vertices 1, 2, and 3 are now in the stack, all their predecessors have been processed, and Eq. (6.2) has been evaluated for these three vertices. *ee* [5] is the next one determined. When vertex 5 is being processed, *ee* [7] is updated to 11. This, however, is not the true value for *ee* [7], since Eq. (6.2) has not been evaluated over all predecessors of 7 (v_4 has not yet been considered). This does not matter, as 7 cannot get stacked until all its predecessors have been processed. *ee* [4] is next updated to 5 and finally to 7. At this point *ee* [4] has been determined, as all the predecessors of 4 have been examined. The values of *ee* [6] and *ee* [7] are next obtained. *ee* [8] is ultimately determined to be 18, the length of a critical path. You may readily verify that when a vertex is put into the stack, its early time has been correctly computed. The insertion of the new statement does not change the asymptotic computing time; it remains $O(e+n)$.

6.5.2.2 Calculation of Late Activity Times

In the backward stage the values of *le* [*i*] are computed using a function analogous to that used in the forward stage. We start with *le* [*n*−1]=*ee* [*n*−1] and use the equation

$$le[j] = \min_{i \in S(j)} \{ le[i] - \text{duration of } <j,i> \} \tag{6.3}$$

where $S(j)$ is the set of vertices adjacent from vertex *j*. The initial values for *le* [*i*] may be set to *ee* [*n*−1]. Basically, Eq. (6.3) says that if <*j*,*i*> is an activity and the latest start time for event *i* is *le* [*i*], then event *j* must occur no later than *le* [*i*] − duration of <*j*,*i*>. Before *le* [*j*] can be computed for some event *j*, the latest event time for all successor events (i.e., events adjacent from *j*) must be computed. Once we have obtained the topological order and *ee* [*n*−1] from the modified version of Program 6.14, we may compute the late event times in reverse toplogical order using the adjacency list of vertex *j* to access the vertices in $S(j)$. This computation is shown below for our example of Figure 6.40(a).

$le[8] = ee[8] = 18$
$le[6] = \min\{le[8] - 2\} = 16$
$le[7] = \min\{le[8] - 4\} = 14$
$le[4] = \min\{le[6] - 9, le[7] - 7\} = 7$
$le[1] = \min\{le[4] - 1\} = 6$
$le[2] = \min\{le[4] - 1\} = 6$

$le[5] = \min\{le[7] - 4\} = 10$
$le[3] = \min\{le[5] - 2\} = 8$
$le[0] = \min\{le[1] - 6, le[2] - 4, le[3] - 5\} = 0$

If the forward stage has already been carried out and a topological ordering of the vertices obtained, then the values of $le[i]$ can be computed directly, using Eq. (6.3), by performing the computations in the reverse topological order. The topological order generated in Figure 6.41(b) is 0, 3, 5, 2, 1, 4, 7, 6, 8. We may compute the values of $le[i]$ in the order 8, 6, 7, 4, 1, 2, 5, 3, 0, as all successors of an event precede that event in this order. In practice, one would usually compute both *ee* and *le*. The procedure would then be to compute *ee* first, using algorithm *topSort*, modified as discussed for the forward stage, and then to compute *le* directly from Eq. (6.3) in reverse topological order.

Using the values of *ee* (Figure 6.41) and of *le* (above), and Eq. (6.1), we may compute the early and late times $e(i)$ and $l(i)$ and the degree of criticality (also called slack) of each task. Figure 6.42 gives the values. The critical activities are a_1, a_4, a_7, a_8, a_{10}, and a_{11}. Deleting all noncritical activities from the network, we get the directed graph or critical network of Figure 6.43. All paths from 0 to 8 in this graph are critical paths, and there are no critical paths in the original network that are not paths in this graph.

activity	early time	late time	slack	critical
	e	l	$l-e$	$l-e=0$
a_1	0	0	0	Yes
a_2	0	2	2	No
a_3	0	3	3	No
a_4	6	6	0	Yes
a_5	4	6	2	No
a_6	5	8	3	No
a_7	7	7	0	Yes
a_8	7	7	0	Yes
a_9	7	10	3	No
a_{10}	16	16	0	Yes
a_{11}	14	14	0	Yes

Figure 6.42: Early, late, and criticality values

As a final remark on activity networks, we note that the function *topSort* detects only directed cycles in the network. There may be other flaws, such as vertices not reachable from the start vertex (Figure 6.44). When a critical-path analysis is carried out on such networks, there will be several vertices with $ee[i] = 0$. Since all activity times are assumed > 0, only the start vertex can have $ee[i] = 0$. Hence, critical-path analysis

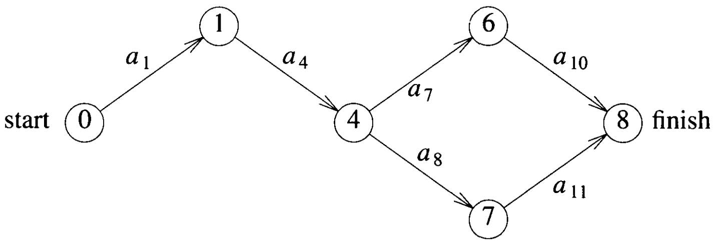


Figure 6.43: Graph obtained after deleting all noncritical activities

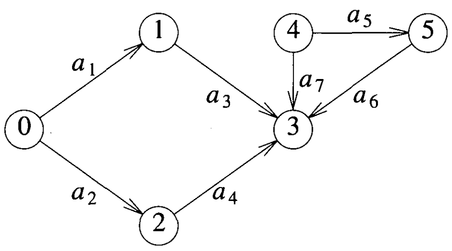


Figure 6.44: AOE network with some nonreachable activities

can also be used to detect this kind of fault in project planning.

EXERCISES

1. Does the following set of precedence relations (<) define a partial order on the elements 0 through 4? Explain your answer.

$$0 < 1;\ 1 < 4;\ 1 < 2;\ 2 < 3;\ 2 < 4;\ 4 < 0$$

2. (a) For the AOE network of Figure 6.46, obtain the *early* and *late* starting times for each activity. Use the forward-backward approach.

   (b) What is the earliest time the project can finish?

   (c) Which activities are critical?