Programming Project – AOE Task Manager

Due date and credits: Friday May 17, 2019 (15% of final grade)

Activity-on-Edge (AOE) Networks

An *activity-on-edge network* or *AOE network* is a directed graph G in which the directed edges represent tasks or activities and the vertices represent events (e.g. completion of some activities). In addition, the directed edges represent precedence relations between tasks.

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 completed. Figure 1(a) is an AOE network for a hypothetical project with 11 tasks or activities: a_1, a_2, …, a_{11}. There are nine events: v_0, v_1, … , v_8. The events v_0 and v_8 may be interpreted as "start project" and "finish project," respectively. Figure 1(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 v_1, v_2, and v_3, respectively, occur. Activities a_7 and a_8 can be carried out concurrently after the occurrence of event v_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 v_4 and v_5 have occurred, a dummy activity a_{12} represented by an edge <5,4> may be introduced.

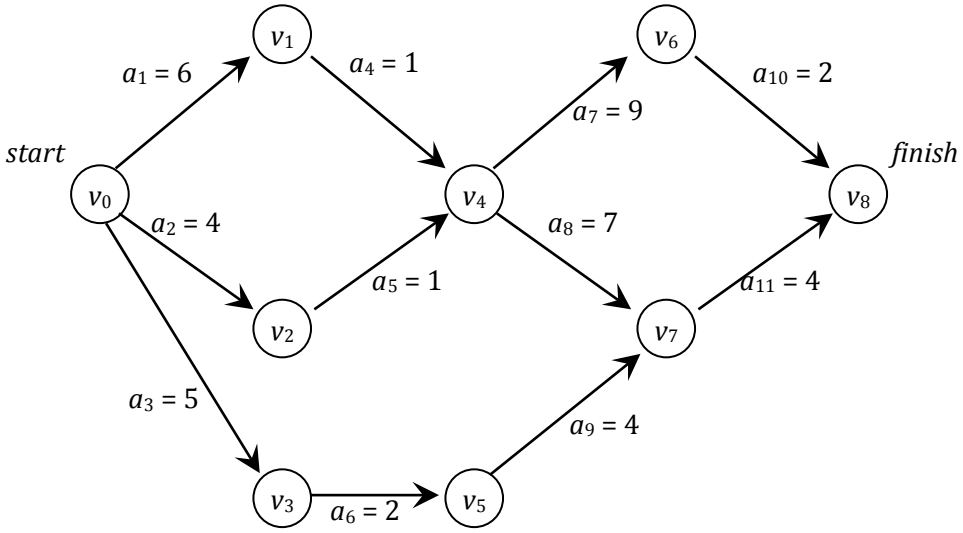

Figure 1(a): Activity Network for a hypothetical project

events	interpretations
v_0	start of project
v_1	completion of activity a_1
v_4	completion of activities a_4 and a_5
v_7	completion of activities a_8 and a_9
v_8	completion of project

Figure 1(b): Interpretation of some of the events in the above activity network

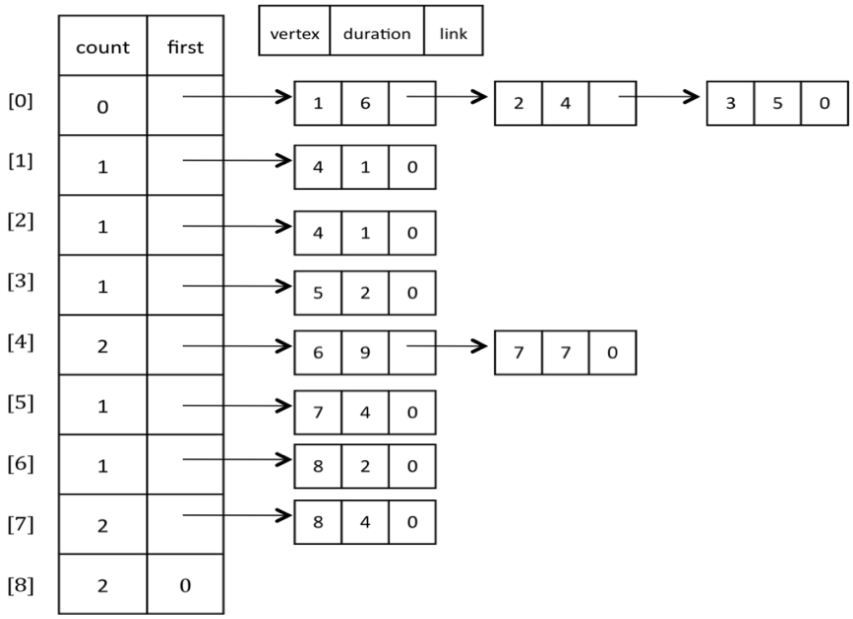

Figure 1(c): Adjacency List for Figure 1(a)

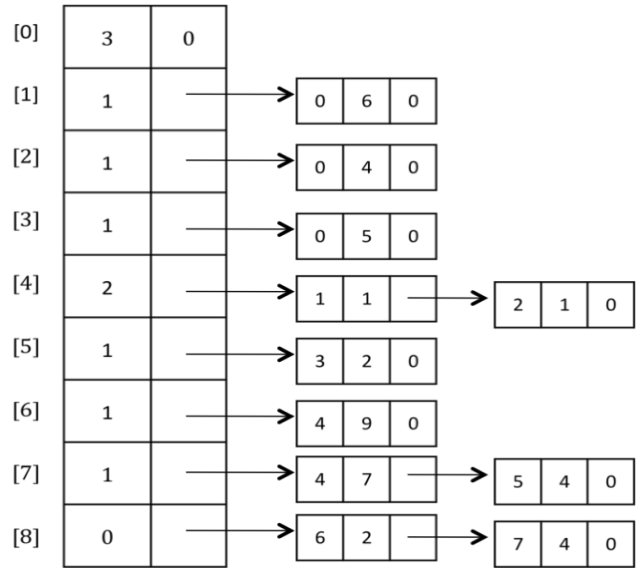

Figure 1(d): Inverse Adjacency List for Figure 1(a) – helpful for Computation of *le* (see Figure 3)

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 some 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 along the vertices 0, 1, 4, 6, 8 is a critical path in the network of Figure 1(a). The length of this critical path is 18. A network may have more than one critical path (the path along the vertices 0, 1, 4, 7, 8 is also critical).

The *earliest time* that an event i (at vertex v_i) can occur is the length of the longest path from the start vertex 0 to vertex i. For example, 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 the earliest start time for activity a_i by $e(i)$. 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 1(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 non-critical 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 1(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. 66.

Finally, 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).

Calculation of Early Activity Times

Recall that events are denoted by vertices. 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(let us call these equations *E&L*):

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

and

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

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 $ee[0]$ and compute the remaining early start times, using the following formula (let us call it equation *EE*):

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

where $P(j)$ is the set of all vertices adjacent (predecessors) to vertex j. If this computation is carried out in *topological order*, the early times of all predecessors of j would have been computed prior to the computation of $ee[j]$.

The algorithm to do this is obtained easily from the *Topological Order Algorithm:*

Forward Stage (Modified *Topological Order* Algorithm)

(1) The array ee is initialized to zero. The evaluation of equation *EE* is computed in parallel with the generation of the topological order. ee(j) is updated each time the ee(i) of one of its predecessors is known, i.e., when I is ready for output.

(2) Define a struct Pair with two public int data members vertex and dur. It is assumed that dur contains the activity duration.

(3) Define HeadNodes to be an array of List<Pair>s.

(4) It is assumed that the array ee is initialized to zero. ee[j] is updated each time the ee[i] of one of its predecessors is known (i.e. when i is ready for output).

```
void ModTopologicalOrder()
// The n vertices of a network are listed in topological order
{
   int top = -1;
   // create a linked stack of vertices with no predecessors
   for (int i = 0; i < n; i++)
      if (count[i] == 0)
      {
         count[i] = top;
         top = i;
      }
   for (i = 0; i < n; i++)
      if (top == -1) {
         cout << "network has a cycle" << endl;
         return;
      }
      else {
         // unstack a vertex
         int j = top;
         top = count[top]

         ListIterator<Pair>li(HeadNodes[j]);
         if (!li.NotNull()) continue;
         Pair p = *li.First();
         while (1) {
            // decrease the count of the successor vertices of j
            int k = p.vertex;
            if (ee[k] < [ee[j] + p.dur)
               ee[k] = ee[j] + p.dur;
            count[k]--;
            if (count[k] == 0) {
               // add vertex k to stack
               count [k] = top;
               top = k;
            }

            if (li.NextNotNull())
               p = *li.Next();  // p is a successor of j
            else break;
         } // end of while
      } // end of else
}
```

To illustrate the working of the modified *Topological Order* algorithm presented above, let us try it out on the AOE network of figure 1(a). See figure 2 below. 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 equation *EE* 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 equation *EE* has not been evaluated over all predecessors of 7 (v_4 has not been considered yet). 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.

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										

Figure 2: Computation of *ee* for AOE network of figure 1(a)

Since the backward stage (second phase) will be described below only in pseudocode, it may be helpful to note here that the above algorithm was designed based on the following pseudocode description:

```
insert startEvent into Stack

while Stack not empty
{
        pop nextEvent from Stack

        for all neighbors of nextEvent Top Event

        if nextNeighbor.predecessorCount == 1
                Push nextNeighbor into Stack

        if  nextNeighbor.NewEearliestStartTime > nextNeighbor.OldEarliestStartTime
                Update nextNeighbor.EarliestStartTime

        nextNeighbor.predecessorCount --
}
```

Calculation of Late Activity Times

In the backward stage the values of $le[i]$ are computed using a function analogous to the forward stage. We start with $le[n\text{-}1] = ee[n\text{-}1]$ and use the following equation, (let us call it equation *LE*):

$$le[j] = \min_{i \in S(j)} \{ le[i] - \text{duration of } \langle j,i \rangle \}$$

where $S(j)$ is the set of vertices adjacent from vertex j.

The initial values for $le[i]$ may set to $ee[n-1]$. Basically, equation *LE* says that if $\langle j,i \rangle$ 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 $\langle j,i \rangle$.

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. These times can be obtained in a manner identical to the computation of the early times by making minor modifications to the forward stage algorithms.

Figure 3 below describes the process on the network of figure 1(a). 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 equation *LE*, by performing the computations in the *reverse topological order*. The topological order generated in Figure 2 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, as discussed for the forward stage, and then to compute *le* directly from equation *LE* in reverse topological order.

Using the values of *ee* (Figure 2) and of *le* (Figure 3), and equation *E&L*, 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 4 below gives the values. The critical activities are a_1, a_4, a_7, a_8, a_{10} and a_{11}.

le	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
initial	18	18	18	18	18	18	18	18	18	[8]
output 8	18	18	18	18	18	18	16	14	18	[7,6]
output 7	18	18	18	18	7	10	16	14	18	[5,6]
output 5	18	18	18	8	7	10	16	14	18	[3,6]
output 3	3	18	18	8	7	10	16	14	18	[6]
output 6	3	18	18	8	7	10	16	14	18	[4]
output 4	3	6	6	8	7	10	16	14	18	[2,1]
output 2	2	6	6	8	7	10	16	14	18	[1]
output 1	0	6	6	8	7	10	16	14	18	[0]

Figure 3: Computation of *le*

As a final remark on activity networks, we note that the algorithms detect only directed cycles in the network. There may be other flaws, such as vertices not reachable from the start vertex. 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 $ee[i]$. Hence, critical-path analysis can also be used to detect this kind of fault in project planning.

Backward Stage

```
set latest event time for all event to project duration

insert last event into Stack

while Stack Not Empty
{
  pop finishEvent from Stack

  for all Neighbor current of Top Event

  if nextNeighbor.successorCount == 1
        Push nextNeighbor into Stack

  If nextNeighbor.NewLatestStartTime < nextNeighbor.OldLatestStartTime
        Update nextNeighbor.LatestStartTime

  current.successorCount --
}
```

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 4: Final Result: early, late, and criticality values

Output

Please see sample input Graph test cases and the corresponding Input & Output text streams.