CS 421 Algorithms (Summer 2018)

Lab Assignment #3: Critical Path Analysis

Due on 6/23/2018, Saturday (11PM)

1. Introduction:

In this assignment, we will do a critical path analysis from the field of project management. The critical path analysis is often used to schedule tasks associated with a project. Normally, tasks of a project forms a directed acyclic graph (DAG). From a DAG, critical path analysis can find a critical path and some other useful information.

2. Background: DAG Representation of a Project

Figure 1 below gives an example of a DAG representation of a project. Each vertex represents an activity that must be completed, along with the time required to complete it. The graph is thus called an *activity-node graph*, in which vertices represent activities and edges represent precedence relationships. An edge (u, v) indicates that activity u must be completed before activity v may begin, which implies that the graph must be acyclic. Any activities that do not depend (either directly or indirectly) on each other can be performed in parallel.

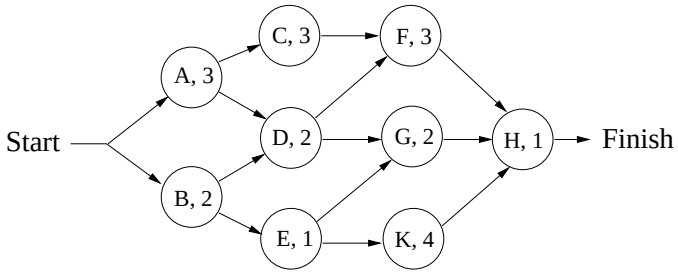

Figure 1: An activity-node graph. The number in each node is the processing time for each activity

There are two important questions must be answered. First, what is the earliest completion time for the project? The answer, as the example graph shows, is 10 time units – required along path A, C, F, H. Second, which activities can be delayed, and by how long, without affecting the minimum completion time? For example, delaying any of A, C, F, H would push the completion time of the project past 10 time units. However, activity B is less critical and can be delayed up to 2 time units.

To perform these calculations, we convert the activity-node graph to an *event-node graph*, in which each event corresponds to the completion of an activity and all its dependent activities. Events reachable from a node u in the event-node graph may not commerce until after the event u is completed. In the graph conversion, dummy edges and vertices may need to be inserted to avoid introducing false dependencies or false lack of dependencies. Figure 2 shows the corresponding event-node graph of the activity-node graph in Figure 1.

Critical path analysis of a project usually are expected to provide the following three useful information to the project manager.

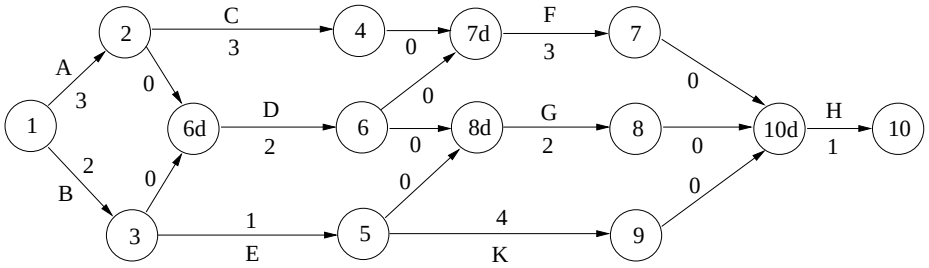

Figure 2: The corresponding event-node graph

- Earliest completion time for each activity (or task):
  Since both graphs are DAGs, to find the earliest completion time of each activity, we merely need to find the length of the *longest* path from the first event to the event corresponding to the completion of the activity in the event-node graph. The shortest path algorithms can be easily adapted to compute the longest path. Let EC_i be the earliest completion time for node i in the event-node graph. The applicable rules are

$$EC_1 = 0 \quad and \quad EC_v = Max_{(u,v)\in E}(EC_u + c(u,v))$$

  where node 1 is the node corresponding to the *start* of the project.

- Latest completion time for each activity (or task):
  We can also compute the latest completion time of each activity without affecting the project earliest completion time. Let LC_i be the latest completion time for node i in the event-node graph. The applicable rules are

$$LC_n = EC_n \quad and \quad LC_u = Min_{(u,v)\in E}(LC_v - c(u,v))$$

  where node n is the node corresponding to the *finish* of the project.

- Slack time for each activity (or task): The slack time of an activity is the amount of time that the completion of the activity can be delayed without delaying the overall project's completion time, or

$$Slack_{(u,v)} = LC_v - EC_u - c(u,v)$$

Figure 3 shows all the EC, LC and $Slack$ values of the given example, where the values of EC are specified above each node, the values of LC are specified below each node, the values of $Slack$ are specified beside each edge within a parenthesis.

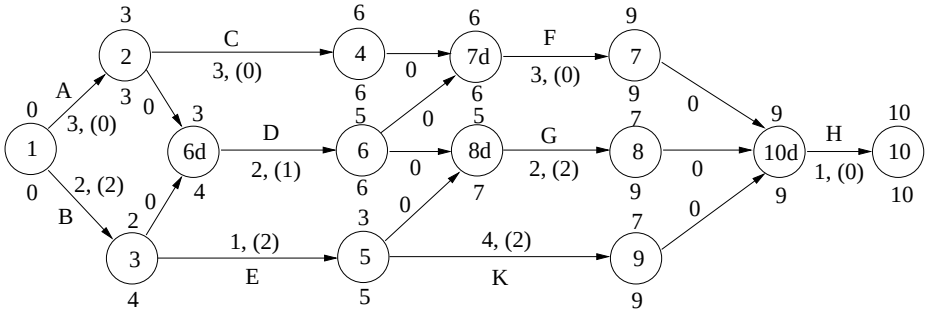

Figure 3: The EC, LC, and Slack values for each activity

3. Design of the Program:

3.1 Input graph

A list of files, containing testing activity-node graphs, will be provided in the following directory.

`/home/JHyeh/cs421/labs/lab3/files/`

Each file has one graph in it. The graph is represented as an `adjacency-matrix` M in the file, where

$$M(i,j) = \begin{cases} \text{a non-negative value} & \text{if the edge } (i,j) \in E \\ -1 & \text{if } (i,j) \notin E \end{cases}$$

The non-negative value in an entry $M(i,j)$ actually denotes the processing time of the node (activity) j. In the matrix, the first row and the first column correspond to the `Start` activity, and the last row and the last column correspond to the `Finish` activity.

Command-line argument: There is only one command-line argument - <file name> containing the input activity-node graph. Your program should read in the input activity-node graph and store the graph using the `adjacency-list` representation in your program. If the graph has n activities (including the two dummy activities, `Start` and `Finish`), an array of size n should be allocated, where there are two entries for the two dummy activities (see Figure 1). The processing time for both `Start` and `Finish` are 0.

3.2 Graph transformation: To transform an activity-node graph G to the corresponding event-node graph G', the following guidelines can be followed.

1. Let $G' = G$.
2. For each node, the weights of all its incoming edges are equal to the processing time of the node.
3. Identify nodes in G' with multiple incoming edges. For each such node u, an `associate` node u' is created in G'. A new edge (u', u) is inserted into G' and all the original incoming edges of u will be redirected to u'. The weight of the new edge (u', u) is equal to the processing time of the node u and the weights of those re-directed edges will be re-set to 0.

3.3 Efficient EC and LC computation: Since the graphs are DAGs, the earliest completion times for nodes can be computed by their topological order, and the latest completion times can be computed by the reverse topological order.

3.4 Output: For each activity, your program should list its EC, LC and Slack times in one output line.

4. Submission:

Use only `Java` to implement this assignment. Before submitting your program, please make sure your program can be compiled and run in `onyx`. Submit your programs from `onyx` by copying all of your files to an empty directory (with no subdirectories) and typing the following FROM WITHIN this directory:

```
submit jhyeh cs421 p3
```