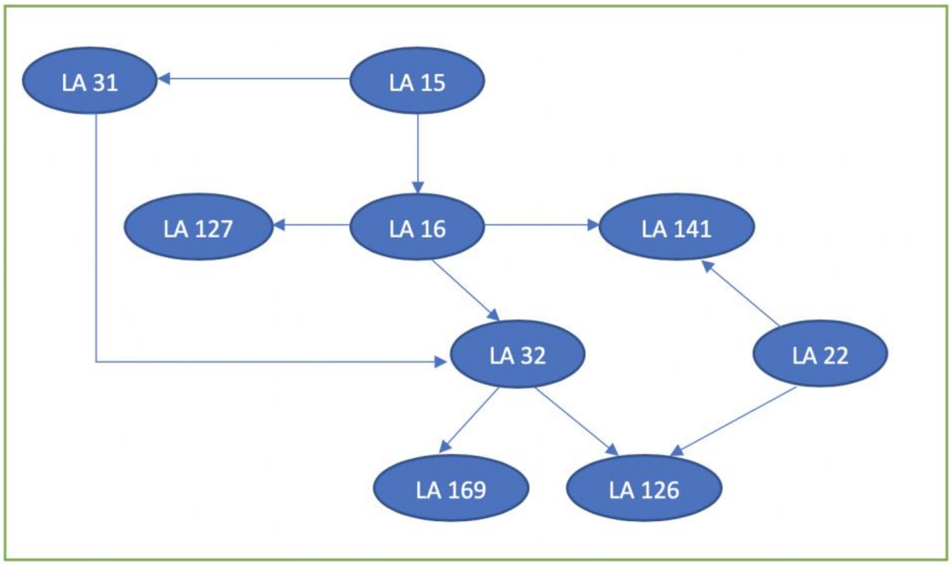
Homework 6 - Solution

R-13.4

The figure below shows Directed Acyclic Graph constructed from the description of the problem.



One of the sequences here will be:
LA15, LA16, LA31, LA32, LA127, LA169, LA22, LA126, LA141

C-13.4

We can use adjacency list representation to represent graph *G*. This will take $O(n + m)$ space. Now, for each vertex, its adjacency list will keep a list of vertices which are reachable from the vertex. We will maintain this list of vertices as a balanced binary search tree. Since we have *n* vertices, maximum length of the adjacency list will be *n*, in the case a vertex is connected to all other vertices of the graph. Thus, to check if vertices *v* and *w* are adjacent, we will use binary search over the adjacency list, which would run in $O(log\ n)$ time.

A-13.4

a. Algorithm to compute a center of *T* of a given *n*-node free tree *T*:
    1. Remove all leaves of the tree *T* and let the remaining tree be T_1.
    2. Now, remove all leaves of T_1 and remaining tree is T_2.
    3. Repeat step 2 (as remove all leave nodes of T_i and the remaining tree) is T_{i+1} till tree has only one or two nodes left.

Let in the last step we are left with remaining tree T_k. Now if T_k has only one node, that is the center of T and k is the eccentricity, and if T_k has two nodes left, either can be the center of T and the eccentricity of the center node is $k+1$. The above algorithm in worst case can take $O(n)$ time.

b. No, not always unique. As we discussed in (a) part, if the last remaining tree has two nodes, then either can be the center of the tree, which means the free tree can have two distinct centers.

C-14.3

If we avoid the use of locator pattern,

a) We can't locate an item in the Priority Queue Q.
b) We can't update the key of an item in the Priority Queue Q.

The main change to the description of Dijkstra's algorithm is that now when we remove a vertex, u, with smallest key, we need to check if u has already been removed earlier in the algorithm. To perform this check, we could add a Boolean flag to each vertex that is true if and only if that vertex has already been processed (and added to the cloud). We simply set flag as false for every vertex during initialization. So, when we remove a vertex, we test if this flag is true. If it is false, then this is the first removal for vertex u, so we do all the edge relaxation operations for u and at the end we set u flag as true. If the flag for u is true, then we just ignore this vertex and repeat the $removeMin()$ operation in the while loop.

The running time for Dijkstra's algorithm is still $O(m \log n)$ in this case, because $O(\log m)$ is $O(\log n)$, and each heapify operation is done once for each edge, just as before.

If we use data structure as min Heap instead of priority queue, the runtime is as follows. Since constructing a bottom up heap would be $O(n)$ & we remove m edges form the queue which would be $O(m \log m)$ and initialization of the queue is $O(n \log m)$. So, in worst case the total running time is $O((n+m)\log m)$.

A-14.1

Form the description of the graph, we can conclude that it is a DAG since

a) The Graph G has n vertices & m edges
b) Moreover, all the edges are directed with cost as $c(e)$ (Negative for prize & Positive for other items).

The minimum cost monotone path from G can be obtained by DAG Shortest Path Algorithm. We use a Depth First Algorithm & get a topological order of the vertices which costs $O(n + m)$. Then Perform edge relaxation in the order of the topological sort which also costs $O(n + m)$.

A-14.4

Here we not only need to find out the shortest path, or shortest time from place A to place B, as there will not be a plane ready to take you there always. We also need to consider the connecting time at the airports. Thus, the basic idea here is using a Dijkstra algorithm combined with modification of edges based on time. We will have an array $D[]$ to store the distance, and a list of queue $R[]$ to store the route, and $T[]$ for the schedule of each flight.

Algorithm flightSchedule(string *a*, string *b*, time *t*):
Input: Origin airport a, Destination airport b, time t to depart
Output: A sequence of airports from a to b

```
$T[a] \leftarrow t$
$R[a].append(a)$
$for\ each\ vertex\ u\ != \ a\ of\ A\ do$
    $T[u] \leftarrow infinite$
    $R[u].append(a)$

$while\ Q\ is\ not\ empty\ do$
    $u \leftarrow Q.removeMin()$
    $currentTime \leftarrow T[u]\ +\ c(u)$
    $if\ u\ =\ b\ then\ do$
        $return\ R[b]$

    $for\ each\ vertex\ z\ adjacent\ to\ u\ such\ that\ z\ is\ in\ A\ do$
        $if\ T[u] +\ w\big((u,z)\big) \leq T[z]\ then$
            $T[z] \leftarrow T[u]\ +\ w((u,z))$
            $change\ the\ key\ for\ vertex\ z\ in\ Q\ to\ T[z]$
            $R[z] \leftarrow append(R[u], z)$
```

The running time analysis is very similar to Dijkstra algorithm. We will be doing m checks for the flight, and $(m+n)$ operations for the priority queue, so the total running time will be $O(m) + O((n+m)\log n)$, which is $O((n+m)\log n)$.