

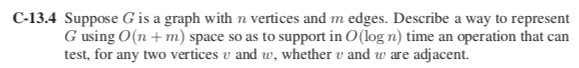
**Solution**:

The sequence of courses that allows Bob to satisfy all the prerequisites

There are many sequences possible, using directed graph.

One possible sequence can be:

LA15, LA16, LA127, LA31, LA32, LA169, LA22, LA126, LA141

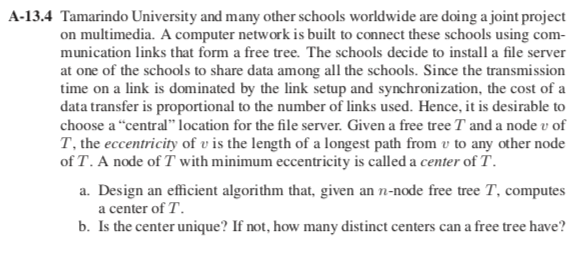


**Solution**:

A way to represent G using *O(n+m)* space is Adjacency List. The total time needed to check all of the neighbors of vertex *v* is proportional to the degree of *v.*

* 1. For each vertex *v* in adjacency list maintains a list which are reachable from *v*.
  2. For *n* vertices, adjacency list will have maximum of length *n* list.
  3. Now sorting the list of neighbors which are reachable from the vertex *v.*
  4. To check the vertices *v* and *w* are adjacent use **binary search** which take time proportional to the logarithm of the degree.

Binary search will take O(log n) time to test any two vertices *v* and *w* are adjacent or not.



**Solution:**

A tree without root is called as free tree. This means that graph does not contain cycle.

Given: A free tree consist of node *v.* The eccentricity of *v* is the longest path from *v* to any other node.

1. Efficient algorithm that computes a center of T.

As T is a tree, there is no cycle in it. We have to remove all the leaf nodes (external nodes) from the tree until a single/two nodes are left in the tree, which will be the center of the tree T.

1. Removing all the leaf nodes (external nodes) of a tree T and name the remaining tree T`.
2. Now, removing all the leaf nodes from T` and name the remaining tree T``.
3. While one or two nodes are left, repeat the above process.
4. If the remaining tree Tk has only one node then it will be the center of the tree T and the eccentricity of the center node will be k.
5. Else for two nodes in tree Tk, the eccentricity of the center node will be k+1.

**Running time**: The running time of the above procedure is O(n). Traversing the tree will take O(n) time and the while loop processes each node only once.

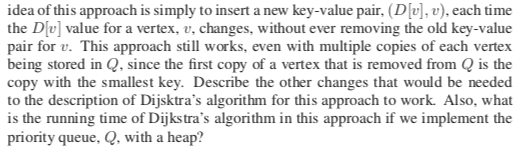
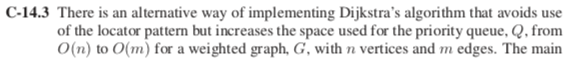
1. No, the center will not be unique.

Two distinct centers for a free tree can be possible.

Consider the longest path P of tree T. The center of the tree T has its path as the median of P.

If length of P is odd, then the center of the tree will be one.

If the length of P is even, then the center of the tree will be two.



**Solution**:

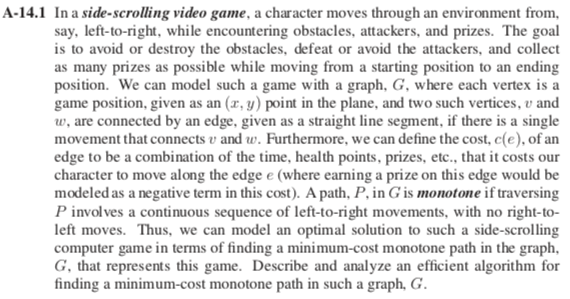
An alternative way to implement Dijkstra algorithm depending on the above condition can be done by maintaining visited nodes in a set.

We can perform three operation INSERT, EXTRACT-MIN, DECREASE-KEY. But

for implementing Dijkstra algorithm using priority queue with heap, Priority queue doesn’t support decrease-key. To resolve this problem, we don’t update the key, but we insert a copy of it. This allows multiple instances of the same vertex.

INSERT operation will perform only once for every vertex. And maintaining set S for visited vertices. Each edge in the adjacency list is examined only once.

**Running time**: The running time of the above algorithm when implement the priority queue Q with a heap will be O(m log n) where n is the number of vertices and m is the edges.



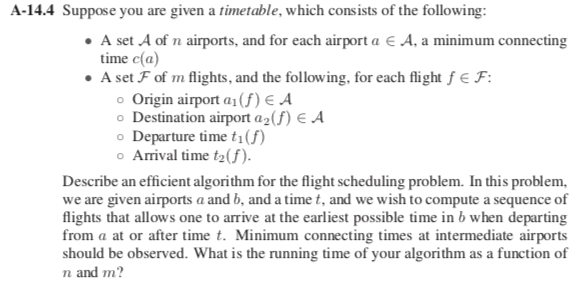
**Solution:**

Given: A path P in a graph G, if traversing P involves a continuous sequence of left-to-right movement only.

The graph formed will be the Directed Acyclic Graph (DAG), we can compute a topological ordering of *n* vertices (game positions) in an *m* edge DAG to calculate single source shortest distance.

We can initialize all vertices as infinite and distance to source as 0. Finding topological ordering of the graph represent as linear ordering. Processing every vertex one by one in topological order, update distances of its adjacent using distance of current vertex.

**Running Time**: The running time of the algorithm is *O(n+m)* where *n* are the vertices and *m* are the edges of DAG G. After finding topological order the algorithm process all the vertices and for every vertex it runs a loop. Considering adjacent vertices in a graph as O(*m*). Inner loop runs for *O(n+m)* Time.



**Solution**:

Given:

If a flight *f* origin from the airport *a1(f)*

Destination airport *a2(f)*

Departure time *t1(f)*

Arrival time *t2(f)*

Naming n airports as *a1, a2, a3………… an* and flights as *f1, f2, f3……… fn*

1. Considering the above situation in terms of di-graph, taking airports as vertices(*n*) and flights as edges(*m*). Edges are flights with two weights.
2. Where minimum connecting time for each airport *a* is *c(a)* which can be represented as corresponding weight of the edges.
3. Suppose *s* is the origin airport and start.s is the starting time. Considering earliest arrival time, T
4. Starting with the vertex *s,* set the starting time T[s] as 0. Now checking for all the vertices(*a*) with arrival time T[*a*].
5. Make priority queue, Q for vertices keyed by T. Checking, while Q is not empty than remove minimum value from the priority queue.
6. Now for all of the vertices say *w* that are adjacent to *a* and present in the Q. Determining the next flight by taking another priority Queue, *Q1*, for flights *f* with departure time *t1(f)* keyed by *t2(f)*
7. If *Q1*, is not empty, then no need to remove minimum value based on the arrival time(*t2(f)*).
8. Relaxing the edges, considering the minimum time taken by the adjacent edges T[w]. If T[w] is less than the time then update w in priority queue Q. after that return the earliest arrival time T.

**Running time**: Using Dijkstra algorithm implementation using priority queue will take the running time *O(m log n)* where *m* are the edges(flights) in the graph and *n* are the vertices(airports).