CS 600 Advanced Algorithms

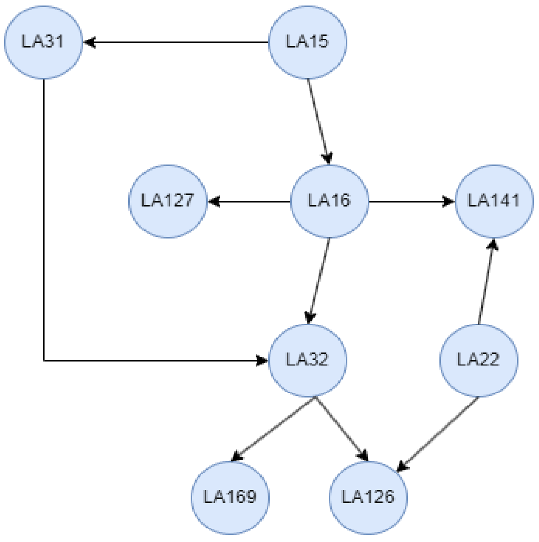
Sri Dhanush Reddy Kondapalli

CWID: 10476150

Homework 6

1 R-13.7.4 Bob loves foreign languages and wants to plan his course schedule to take the following nine language courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA169. Find a sequence of courses that allows Bob to satisfy all the prerequisites.

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

2 C-13.7.19 Suppose G is a graph with n vertices and m edges. Describe a way to represent G using O(n+m) space so as to support in O(log n) time an operation that can test, for any two vertices v and w, whether v and w are adjacent.

For graph G and two vertices v and w:

1. Begin by locating the linked component in the graph.
2. Provide a number to each vertex, as well as the start and finish times of each vertex.
3. Stop parsing when both vertices v and w are covered by the parser.
4. The compiler will take log n time till step 3. Where n is the number of graph vertices.
5. When the compiler reaches the second vertices after step 4, it verifies the prior vertex. If the preceding vertex is the w vertex's v. It indicates that two vertices are adjacent.
   Space Complexity: To store the n vertices and m edges will take
   O (n + m)
   Time Complexity: To delete an edge from the graph using same procedure will take log (n) time.

3 A-13.7.37 Given a free tree T and a node v of T, the eccentricity of v is the length of a longest path from v to any other node of T. A node of T with minimum eccentricity is called a center of T.

a) Algorithm for computing the center of T of an n-node free tree T:

1. Remove all leaves of T. Let the remaining tree be T_1
2. Remove all leaves of T_1. Let the remaining tree be T_2.
3. Repeat the remove operation as follows: Remove all leaves of T_i. Let remaining tree be T_{i+1}.
4. Once the remaining tree has only one node or two nodes, stop Suppose now the remaining tree is T_k.

5. If T_k has only one node, that is the centre of T. The eccentricity of the centre node is k.
6. If T_k has two nodes, either can be the centre of T. The eccentricity of the centre node is k + 1.

b) No, it is not always unique. As we saw in (a), if the final surviving tree contains two nodes, any of them can be the tree's center, implying that the free tree can have two separate centers.

4 C-14.7.11 Describe the other changes that would be needed to the description of Dijsktra's algorithm for this approach to work. Also, what is the running time of Dijkstra's algorithm in this approach if we implement the priority queue, Q, with a heap?

The significant difference in the explanation of Dijkstra's method is that now, when we remove a vertex, u, with the smallest key, we must check to see if u was previously removed in the procedure. We might add a Boolean flag to each vertex that is true if and only if that vertex has previously been processed to do this check (and added to the cloud). During startup, we just set the flag to false for each vertex. When we delete a vertex, we check to see if this flag is true. If it is false, then this is the first removal for vertex u, thus we do all edge relaxation operations on u before setting the u flag to true. If it is false, then this is the first removal for vertex u, thus we do all edge relaxation operations on u before setting the u flag to true. If the u flag is true, we simply ignore this vertex and repeat the removeMin() procedure in the while loop.

Dijkstra's method still runs in O(m log n) time since O(log m) = O(log n) and each heapify operation is performed once for each edge, as before.

The run-time is as follows if we use a data structure called min Heap instead of a priority queue. Because building a bottom-up heap is O(n), removing m edges from the queue is O(m log m), and initializing the queue is O(n log m). In the worst-case scenario, the total running time is O((n + m) log m).

5 A-14.7.17 Describe and analyze an efficient algorithm for finding a minimum-cost monotone path in such a graph, G.

The shortest monotonic route. Find a monotonic most restricted path from s to each other vertex in an edge-weighted digraph. A path is monotonic if the heaviness of each edge on the path is carefully growing or decreasing.

Dijkstra's algorithm might be adapted to tackle this problem. The important concept is that relaxing should be performed in the priority queue rather than at each graph node.

Here is a list of changes to the standard Dijkstra's algorithm. I only evaluate edge relaxation in ascending order, resulting in a strictly decreasing shortest path (to obtain an increasing shortest path, change items 2 and 4):

1. Order outgoing edges by weight
2. Each node should have a location in the list of incoming edges (initialized by position of the lightest edge).
3. Priority queue is not required to allow "reduce" operations (so it could be implemented by simple min-heap). Each vertex is placed in the priority queue and is never changed until it reaches the top of the list (as a result each vertex may be represented in the queue several times). Queue entry consists of a key (path length, as usual), vertex, and incoming edge weight. As a result, we may assume that the priority queue contains incoming edges rather than vertices.
4. Relaxation procedure: remove the edge (and thus the vertex where this edge ends) from the queue; for all outgoing edges of the vertex in increasing order, beginning with the position stored in the graph node and ending when the outgoing edge's weight is greater or equal to the incoming edge's weight, push the outgoing edge to the priority queue and advance the stored position.

This approach guarantees that each edge is handled just once (or twice if both strictly decreasing and strictly rising pathways are considered), hence its complexity is O(E log E).

6 A-14.7.20 Describe an efficient algorithm for the flight scheduling problem. In this problem, we are given airports a and b, and a time t, and we wish to compute a sequence of flights that allows one to arrive at the earliest possible time in b when departing from a at or after time t. Minimum connecting times at intermediate airports should be observed. What is the running time of your algorithm as a function of n and m?

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

while Q is not empty do
     u=Q.removeMin()
     currentTime=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((u,z)) <= T[z] then
               T[z] = T[u] + w((u,z))
               change the key for vertex z in Q to T[z]
               R[z]=append(R[u],z)
```

The Dijkstra algorithm is quite similar to the running time analysis. We will do m flight checks and (m + n) priority queue operations, therefore the overall running time will be O(m) + O((n + m) log n), which is O((n + m) log n).