Homework – 6

Jay Kalyanbhai Savani

CWID – 20009207

Graphical user interface, text, application

Description automatically generated

Ans: A prerequisite is a thing/ activity required as a prior condition for something else to happen or exist.

The sequence of courses that would allow Bob to satisfy their courses is as listed below:-

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

Graphical user interface, text, application, email

Description automatically generated

Ans : Input: A graph G with 2 vertices v and w

* Step 1: Begin by finding the connected component in the graph
* Step 2: Number each vertex. Like, start time/ end time of each vertex
* Step 3: Once when both our vertices, v and w, are covered by the parser, stop the parsing. Till here the algorithm costs us O(log n) time
* Step 4: Now we reach the 2nd vertex, and it checks for the previous vertex. If the previous one is a v of w vertex – it means both our vertices are adjacent. Now this takes only O(1) time
* The total time taken for our compilation is O(log n) + O(1) = O(log n) time.
* Deleting an edge from the graph takes O(log n) time, considering we use the same procedure.
* And to store n vertices, and m edges, it costs us O(n+m) time

Graphical user interface, text, application, email

Description automatically generated

Ans: Part (a):

* Step 1: Firstly, we remove the leaves of Tree T. Let the remaining tree be T1.
* Step 2: Now, we remove the leaves of Tree T1. This gets us to T2.
* Step 3: We consequently perform this action repeatedly to get rid of all the leaves from Ti, and let our tree remain like Ti+1.
* Step 4: We stop when our remaining tree has only one/ two nodes left. Let our final tree be Tk.
* Step 5: Say Tk has only one node left, then that is the center of T. Now, the eccentricity of the center node is k.
* Step 6: Say Tk has 2 nodes left, then either of them can be the center of T. And the eccentricity of the center node is k+1.

Part (b):

* The center is not necessarily always unique.
* The chances are the remaining tree has only 2 nodes.
* In situations of a 2-node tree, we don’t remove any leaves.
* In our below diagrammatic examples:
  + The first example can either have D or E as the center, wherein the eccentricity = 3.
  + The second example has center as E, wherein the eccentricity = 4

Background pattern

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Ans:

* Our two calculations seem to ensure the equivalent shortest paths weight.
* Dijkstra’s pick the shortest path as indicated by the greedy procedure.
* Bellman-Ford would pick the shortest path based on the request of relaxations.
* However, these 2 shortest path trees might be unique
* No – There seems to exist a 0-weight cycle.
* At this point it is conceivable that loosening up an edge can probably wreck the parent pointers with the goal that is difficult to reproduce the path back to our source node.
* The simplest instance: we have a vertex v, with a 0-weight edge indicating back to itself.
* If we loosen up the edge, v’s parent pointer in will indicate back to itself.
* When we reproduce a path from a vertex back to the source – we experience v – wherein we get stuck and cannot move forward.
* Deploying the priority queue using the heap – will take O(1) time for getting the max value and O(log n) to sort using the heap.

Graphical user interface, text, application

Description automatically generated

Ans: Monotonic shortest path:

* Let’s say we have an edge weight digraph, we have to now locate monotonic, most limited way from s to each other vertex.
* A way is defined monotonic if the heaviness of our edge is either expanding or diminishing
* Our issue can be address by making some twitches to Dijkstra’s algorithm.
* Primarily, the relaxation should not be performed with min operation at every graph node, but instead in a priority queue.
* We can perform following modifications on Dijkstra’s
  + Ordering our outgoing edges by weight
  + Every node contains the position in the list of outgoing edges
  + There is minimal to no need for our PQ (Priority Queue) to support the decrease operation.
  + So, each vertex is entered in the PQ and never changed until it shows up at the top of the queue.
  + Queue entry consists of key, vertex, and the weight of our incoming edge – so, safe to assume PQ to contain incoming edges rather than vertices.
  + Relaxation Process:
    - Popping the edge from the queue.
    - For all outgoing edges of the vertex in an increasing order, based on the position where they are stored in the graph.
    - Ending when the weight of the outgoing edge >= the weight of the incoming edge
    - Push outgoing edge to the PQ

This approach assures us that each edge is processed at max, once.

Time complexity: O(E log E)

Graphical user interface, text, application, email

Description automatically generated

Ans: We can convert our problem into a graph traversal problem like below:

* Visualize every as a node in the graph
* Each flight makes it in an edge of the graph
* Consequently, the flight time corresponds to the weight of the edge
* A probable limitation: difference between the time of the next departing flight the arrival of previous flight should be high.
* This will reduce our number of edges, so we traverse through the graph to find out an optimum solution.
* Now if you carefully notice, our concern in the time optimization since this has now become a graph traversal problem.
* This can be achieved by depth first search.
* This traversal ends once we find the set of paths from source node/ airport to destination node/ airport and report the cost associated with such paths.
* We need to track our cycles to identify and eliminate if a node has already been visited.
* Like our traditional DFS, the cost of time will O(n+m), where
  + n = number of airports/ nodes
  + m = number of flights/ edges