Homework – 6
Jay Kalyanbhai Savani
CWID – 20009207

EXERCISE | 13.7.4

(a) 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. The course prerequisites are:
- LA15: (none)
- LA16: LA15
- LA22: (none)
- LA31: LA15
- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32.

Find a sequence of courses that allows Bob to satisfy all the prerequisites.

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

EXERCISE | 13.7.19

(a) 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.

Feedback?

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 2^{nd} 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

EXERCISE | 13.7.37

Tamarindo University and many other schools worldwide are doing a joint project on multimedia. A computer network is built to connect these schools using communication links that form a free tree. The schools decide to install a file server at one of the schools to share data among all the schools. Since the transmission time on a link is dominated by the link setup and synchronization, the cost of a data transfer is proportional to the number of links used. Hence, it is desirable to choose a "central" location for the file server. 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) Design an efficient algorithm that, given an n-node free tree T, computes a center of T.

(b) Is the center unique? If not, how many distinct centers can a free tree have?

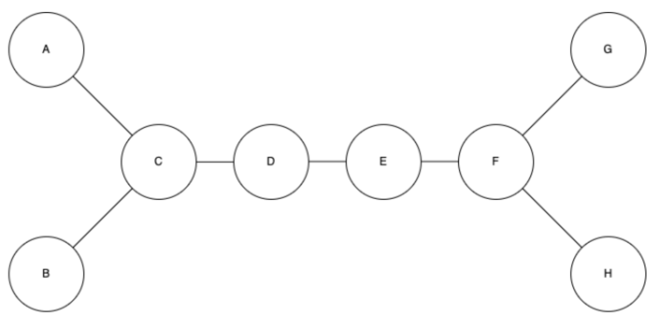
Feedback?

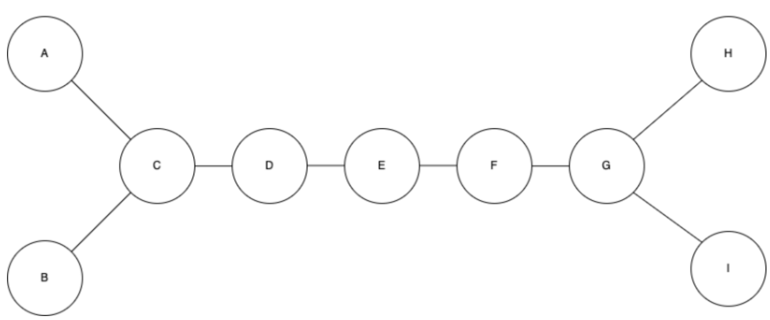
Ans: Part (a):

- Step 1: Firstly, we remove the leaves of Tree T. Let the remaining tree be T_1.
- Step 2: Now, we remove the leaves of Tree T_1. This gets us to T_2.
- Step 3: We consequently perform this action repeatedly to get rid of all the leaves from T_i, and let our tree remain like T_{i+1}.
- Step 4: We stop when our remaining tree has only one/ two nodes left. Let our final tree be T_k.
- Step 5: Say T_k has only one node left, then that is the center of T. Now, the eccentricity of the center node is k.
- Step 6: Say T_k 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





EXERCISE | 14.7.11

(a) There is an alternative way of implementing Dijkstra's algorithm that avoids use of the locator pattern but increases the space used for the priority queue, Q, from $O(n)$ to $O(m)$ for a weighted graph, G, with n vertices and m edges. The main idea of this approach is simply to insert a new key-value pair, $(D[v], v)$, each time the $D[v]$ value for a vertex, v, changes, without ever removing the old key-value pair for v. This approach still works, even with multiple copies of each vertex being stored in Q, since the first copy of a vertex that is removed from Q is the copy with the smallest key. 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?

Feedback?

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.

EXERCISE | 14.7.17

(a) In a ***side-scrolling video game***, a character moves through an environment from, say, left-to-right, while encountering obstacles, attackers, and prizes. The goal is to avoid or destroy the obstacles, defeat or avoid the attackers, and collect as many prizes as possible while moving from a starting position to an ending position. We can model such a game with a graph, G, where each vertex is a game position, given as an (x, y) point in the plane, and two such vertices, v and w, are connected by an edge, given as a straight line segment, if there is a single movement that connects v and w. Furthermore, we can define the cost, $c(e)$, of an edge to be a combination of the time, health points, prizes, etc., that it costs our character to move along the edge e (where earning a prize on this edge would be modeled as a negative term in this cost). A path, P, in G is ***monotone*** if traversing P involves a continuous sequence of left-to-right movements, with no right-to-left moves. Thus, we can model an optimal solution to such a side-scrolling computer game in terms of finding a minimum-cost monotone path in the graph, G, that represents this game. Describe and analyze an efficient algorithm for finding a minimum-cost monotone path in such a graph, G.

Feedback?

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)

EXERCISE 14.7.20

(a) Suppose you are given a *timetable*, which consists of the following:

- A set $\mathcal{A}$ of n airports, and for each airport $a \in \mathcal{A}$, a minimum connecting time $c(a)$
- A set $\mathcal{F}$ of m flights, and the following, for each flight $f \in \mathcal{F}$:
  - Origin airport $a_1(f) \in \mathcal{A}$
  - Destination airport $a_2(f) \in \mathcal{A}$
  - Departure time $t_1(f)$
  - Arrival time $t_2(f)$.

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?

Feedback?

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