Romil V. Shah
20008692

CS / CPE 600
Prof. Reza Peyrovian

Homework Assignment 6.
Submission Date: 10/30/2022

Q1. (No. 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.

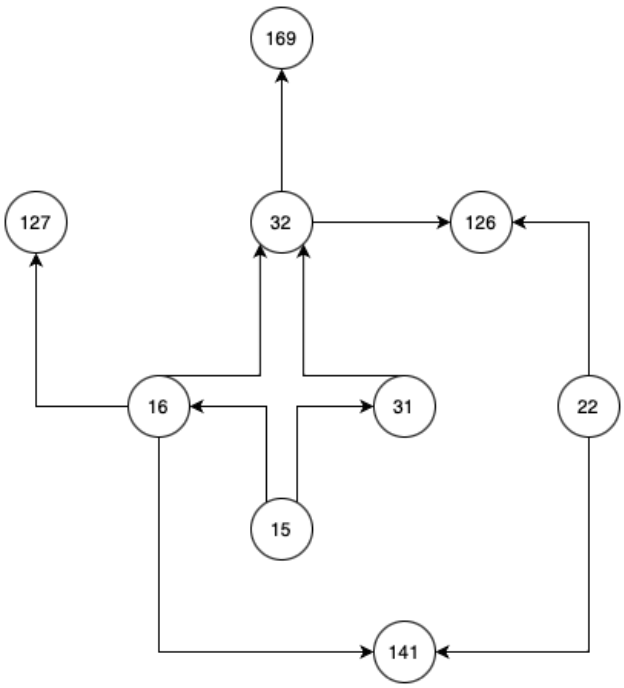
Sol.

Using directed graph, we can design many different possibilities, so that Bob can study all the courses by satisfying all the prerequisites.

Sequence 1:
LA15, LA22, LA16, LA31, LA127, LA32, LA141, LA169, LA126

Sequence 2:
LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, LA169

Sequence 3:
LA15, LA16, LA127, LA31, LA32, LA169, LA22, LA126, LA141



Q2. (No. 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.

Sol.

A way to represent graph G using O(n + m) space is Adjacency List.

An adjacency list is a collection of unordered lists used to represent a finite graph. Its representation for a graph associates each vertex in the graph with the collection of its neighboring vertices or edges. It represents a graph as an array of linked lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The total time needed to check all the neighbors of vertex ‘v’ is proportional to the degree of ‘v’.

a. For each vertex, ‘v’ in adjacency list maintains a list which are reachable from ‘v’.
b. For ‘n’ vertices, adjacency list will be of length ‘n’.
c. Now, we sort the list of neighbors which are reachable from the vertex ‘v’.
d. To check that the vertices ‘v’ and ‘w’ are adjacent, we will use binary search which will take time equivalent to the logarithm of the degree.

Here, binary search will take O(log n) time to test any two vertices ‘v’ and ‘w’ are adjacent or not. So, the run time of this test operation will be O(log n).

Q3. (No. 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?

Sol.

A tree without a designated root node is known as a free tree (Unrooted Tree). It means that the graph does not contain cycle.

Given that, a free tree consists of node v, the eccentricity of v is the longest path from v to any other node.

a. Efficient Algorithm to compute center of T:

As T is a tree, there is no cycle in it. We remove all the leaf nodes from tree T until a single or 2 nodes are left in the tree, which will be the center of Tree T.

1. We start by removing the leaf nodes of the tree T and rename the remaining tree ‘T`’.
2. Now, we remove all the leaf nodes of the tree ‘T`’ and rename the remaining tree ‘T``’.
3. Until 1 or 2 nodes are left, we keep repeating the above process.
4. If the remaining tree T^k 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. Otherwise, if there are 2 nodes in tree T^k, the eccentricity of the center node will be k+1.

Traversing the tree will take O(n) time and the while loop processes each node only once. So, the algorithm will take O(n) time to run.

b. No, the center won’t be unique, as two distinct centers for a free tree are possible.

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

Q4. (No. 13.7.11)

(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?

Sol.

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

We can perform three operations 'Insert', 'Extract-Min', 'Decrease-Key'. But priority queue doesn't support 'Decrease-Key' operation. So, to resolve this problem, we don't update the key and rather insert a copy of it. This will allow us to insert multiple instances of the same vertex.

'Insert' operation will perform only once for every vertex and will maintain set S for visited vertices. Each edge in the adjacency list will be examined only once.

When we implement the priority queue Q with a heap it will take O(m log n) time, where n is the number of vertices and m is the edges.

Q5. (No. 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.

Sol.

Given that, a path P in 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). The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.

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. Then, we find topological ordering of the graph and represent as linear ordering. We process every vertex one by one in topological order, update distances of its adjacent using distance of current vertex.

After finding topological order the algorithm processes all the vertices and for very vertex it runs a loop. Considering adjacent vertices in a graph as O(m), the inner loop runs for O(n + m) time.

So, the run time of the algorithm will be O(n + m) where n are the vertices and m are the edges of Directed Acyclic Graph.

Q6. (No. 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?

Sol.

Given that,
A flight f origin from the airport $a_1(f)$
Destination airport $a_2(f)$
Departure time $t_1(f)$
Arrival time $t_2(f)$

Naming n airports as $a_1, a_2, a_3 \ldots\ldots a_n$ and flights as $f_1, f_2, f_3 \ldots\ldots f_n$

a. 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.
b. Here, minimum connecting time for each airport a is c(a) which can be represented as corresponding weight of the edges.
c. Suppose s is the origin airport and start. s is the starting time. Consider the earliest arrival time, T
d. Starting with the vertex s, set the starting time T[s] as 0. Now, we check for all the vertices(a) with arrival time T[a].
e. Make priority queue, Q for vertices keyed by T. We check while Q is not empty than remove minimum value from the priority queue.
f. Now for all 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 $t_1(f)$ keyed by $t_2(f)$
g. If Q1, is not empty, then no need to remove minimum value based on the arrival time($t_2(f)$).
h. 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.

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).