Sample Solution to Problem Set 3

1. (10 points) Problem 3.16 of text.

Answer. First calculate the indegree $d_{in}(u)$, which is the number of edges into u, for each node u in the prerequisite graph G. This process could be completed in $O(|V|+|E|)$, since we only need to traverse all the nodes in the graph.

According to the problem description, the prerequisite graph can be described as a DAG. By the property in Page 101 of the text, every DAG has at least one source and at least one sink. This suggests an approach to linearization: find a source, output it, and delete it from the graph. Repeat until the graph is empty. Inspired by this idea we can use topological sorting to solve this problem as follows:

1) First we put the nodes whose indegrees are 0 into the first queue, supposed denoted as Q_1.
2) Then decrement the indegrees of all the nodes adjencent to those in Q_1. If the indegrees of such nodes decrement to 0, put them into the second queue, denoted as Q_2.
3) Similarly, decrement the indegrees of all the nodes adjencent to those in Q_2. If the indegrees of such nodes decrement to 0, put them into the third queue, denoted as Q_3.
4) We repeatedly put the 0-indegree nodes into each queue, thus Q_i is just the courses that should be taken in ith semester.

Thus the number of queues is the minimum semesters required by the student to complete all the courses.

2. (10 points) One-way connected graphs

Recall that a directed graph G is strongly connected if for any two vertices u and v, there is a path from u to v *and* there is a path from v to u. We say G is *one-way connected* if for any two vertices u and v, either there is a path from u to v *or* there is a path from v to u (or both).

For instance, the graph over 3 vertices u, v, w and two edges $u \to v$ and $v \to w$ is one-way connected. On the other hand, the graph over 4 vertices u, v, w, x with four edges $u \to v$, $u \to w$, $v \to x$, and $w \to x$ is not one-way connected since neither there is a path from v to w nor there is a path from w to v.

Design a linear-time algorithm for determining whether a given directed graph G is one-way connected.

Answer. According to Property in Page 102 of text, every directed graph is a DAG of its strongly connected components (SCC). So similar to the strongly-connectivity algorithm, we first utilize the

decomposition algorithm in Section 3.4.2 to convert a directed graph into its strongly connected components.

Since we know all the vertices in each SCC are *one-way connected*, the original graph is *one-way connected* if and only if it contains a linear chain in the DAG of its strongly connected components. So we will use topological sorting to order the vertices so as to check if there is a linear chain in the DAG.

More detailed steps can be shown as follows:

1) Run depth-first search on G^R.

2) Run the undirected connected components algorithm on G, and during the depth-first search, process the vertices in decreasing order of their post numbers from step 1.

3) Topologically sort the component graph. Assuming that there are k SCCs, the topological sort gives a linear ordering $< v_1, v_2, ..., v_k >$ of the vertices.

4) Verify that the sequence of vertices $< v_1, v_2, ..., v_k >$ given by topological sort forms a linear chain in the component graph. That is, verify that the edges (v_1, v_2), (v_2, v_3), ... , (v_{k-1}, v_k) exist in the component graph. If the vertices form a linear chain, then the original graph is *one-way connected*; otherwise it is not.

The first two steps are costing $O(|V|+|E|)$ by the proof from the textbook. The third step will also take $O(|V| + |E|)$, because the component graph has at most $|V|$ vertices and at most $|E|$ edges. In the last step we need only check the adjacency list of each vertex v_i in the component graph to verify that there is an edge (v_{i-1}, v_i). We'll go through each adjacency list once.

To sum up, this algorithm will take $O(|V| + |E|)$ time which is linear in the input of the problem.

3. (10 points) Network forensics

You are the chief network administrator of a large corporate network, and have just been informed of a major virus infection in the network. The entire network has been shut down to prevent further damage. You and your forensics team get down to business and immediately start poring over the network logs to figure out what happened and when. Your network consists of n computers which we label C_1, C_2, through C_n. You are able to collect all communication information in the form of m triples of the form (C_i, C_j, t), which means that C_i and C_j communicated with each other at time t.

You know that the virus first infected computer C_1 from an external source at time 0. From then on, there were no further infections from outside the network. But the virus is very infectious. So if C_i communicated with C_j at time t, and if one of them is infected prior to the communication, then the other will certainly be infected at time t.

Given the set of all communication triples, you would like to determine (a) which computers have been infected at the current time, (b) the precise times at which these computers were first infected, and (c) for each infected computer, the path by which the infection first reached the computer. Design an efficient (time polynomial in n and m) algorithm to solve this problem, and analyze its worst-case running time. You may assume for convenience that all times listed in the communication triples are integers.

Answer: Suppose all the triples (C_i, C_j, t) are ordered by t. If not, we can always do this by an $O(m \log m)$ time sort.

(a) We will traverse all the triples with the time t prior to current time. Before the traversal, we define three marks for each computer:

1) INFECTED$[i]$ represents whether computer C_i is infected or not(say 1 for infected and 0 for not), initially all the INFECTED$[i]$ are 0 except computer C_1.
2) INFBY$[i]$ represents computer C_i is infected by which computer(the value is the index of the computer which infects C_i), initially all the INFBY$[i]$ are -1 which means they are not infected.
3) INFTIME$[i]$ represents when computer C_i is infected by others, initially all the INFTIME$[i]$ are -1 which means they are not infected.

We will use the following algorithm to traverse the triples prior to current time.

for all triples (C_i, C_j, t) **do**
  if (INFECTED$[i] == 1$ and INFECTED$[j] == 1$) or (INFECTED$[i] == 0$ and INFECTED$[j] == 0$) **then**
    Continue
  end if
  if (INFECTED$[i] == 1$ and INFECTED$[j] == 0$) **then**
    INFECTED$[j] = 1$; INFBY$[j] = i$; INFTIME$[j] = t$
  end if
  if (INFECTED$[i] == 0$ and INFECTED$[j] == 1$) **then**
    INFECTED$[i] = 1$; INFBY$[i] = j$; INFTIME$[i] = t$
  end if
end for

So when we finish traversal, the computers whose INFECTED$[i]$ is 1, are those that have been infected at the current time.

Since we only need to traverse all m triples in the worst case, and then traverse the INFECTED$[i]$, our algorithm will take $O(m + n)$.

(b) We will do the same traversal as problem (a), for all the triples with the time t prior to current time. To output the result, for each computer C_i, we do the following.

If INFTIME$[i] = -1$, it means this computer has not been infected by this time;

Else the value INFTIME$[i]$ is just the precise times at which these computers were first infected.

Similarly our algorithm only need to traverse all m triples in the worst case, and then traverse the INFTIME$[i]$, our algorithm will take $O(m + n)$.

(c) Similarly after the same traversal as problem (a), for all the triples with the time t prior to current time. For each infected computer C_i, we start from INFBY$[i]$ to find by which it is infected, then recursively we find the infector by reading INFBY array, until we get the computer C_1. For each infected computer, in the worst case, we take $O(n)$ to find the whole infection path.

Thus to find all the infection paths for each infected computer, we take $O(n^2)$ time.

4. (10 points) Problem 4.13 of text.

Answer.

(a) Suppose your car's fuel tank capacity is L. We modify the original graph G to G' as follows:

1) Keep all the routes less than L as the same.
2) Remove all the routes greater than L.

Then we will do a DFS or BFS in G' from the starting city s. If s is connected with t in G', we say there is a feasible way from in s to t; otherwise there is not.

Since the only difference of the above algorithm is we remove all the routes greater than L, which takes at most $|E|$. So this algorithm will still take $O(|V| + |E|)$ time which is linear to the input of the problem.

(b) The problem of finding the minimum fuel tank capacity that is needed to travel from s to t is equivalent to the following problem: Find a path from s to t whose longest edge is minimum, among all paths from s to t. This is because we only need to find one path from s to t, if our fuel can cover the longest route in this path, we can arrive at the destination. This problem is also sometimes referred to as *bottleneck shortest paths* problem.

So we will modify the corresponding part in Dijkstra's algorithm as follows. For each vertex x, we maintain fuel(x) similar to dist(x) in the original algorithm. We initialize fuel(s) to 0 and fuel(x) to ∞ for $x \neq s$.

for all edges $(u, v) \in E$ **do**
  if fuel(v) $> max$(fuel(u), $l(u, v)$) **then**
    fuel(v) = max(fuel(u), $l(u, v)$)
  end if
end for

The above algorithm is correct based on the following fact. If the minimum fuel required in the path $s \to \ldots \to r \to t$ is l, the minimum fuel required in the path $s \to \ldots \to r$ must be no more than l. In this way the modified Dijkstra algorithm can always find the minimum value of the longest routes in each path from s to any node by following the correct order step by step. This value is just the minimum fuel tank capacity that is needed to travel from s to t.

According to the result in Section 4.4.3, we can say the overall running time is $O((|V| + |E|)log|V|)$.