# CSI 604 – Spring 2016

## Solutions to Homework – VI

**Problem 1:**

**Idea:** Suppose the Graph $G(V, E)$ has a spanning tree $T$ such that each node in $L$ is a leaf (i.e., a node of degree 1). Thus, when the nodes in $L$ are deleted from $T$, the remaining graph is a *tree* on the set of nodes $V - L$. Further, for each node $v \in L$, there is a node $w \in V - L$ such that the edge $\{v, w\}$ is in $E$. As the following lemma shows, these two conditions which are necessary for the existence of such a spanning tree are also sufficient.

**Lemma 1:** Let $G(V, E)$ be a connected undirected graph and let $L$ be a subset of nodes. $G$ has a spanning tree in which all the nodes in $L$ appear as leaves if and only if *both* of the following conditions hold: (i) graph $G'(V - L, E')$ obtained from $G$ by deleting the nodes in $L$ (and the edges incident on those nodes) is *connected* and (ii) for each node $v \in L$, there is a node $w \in V - L$ such that the edge $\{v, w\}$ is in $E$.

**Proof:**

**Part 1:** Suppose $G$ satisfies Conditions (i) and (ii) of Lemma 1. We can construct a spanning tree $T$ for $G$ where all the nodes in $L$ are leaves of $T$ as follows. First construct a spanning tree $T'$ for $G'(V - L, E')$. ($T'$ exists since $G'$ is connected.) For each node $v \in L$, by Condition (ii), there is a node $w \in V - L$ such that $\{v, w\}$ is an edge of $G$. Thus, we can attach $v$ to $T'$ as a child of $w$, thus ensuring that $v$ is a leaf in the resulting spanning tree. We get the required spanning tree $T$ after all the nodes in $L$ get attached to appropriate nodes of $T'$.

**Part 2:** Suppose $G$ has a spanning tree $T$ such that each node in $L$ is leaf in $T$. Thus, deleting the nodes in $L$ cannot disconnect the tree. Therefore, the graph $G'(V - L, E')$ has a spanning tree. In other words, Condition (i) holds. The fact that each node $v$ in $L$ is a leaf in $T$ points that the parent $w$ of $v$ is a node in $V - L$; that is, $\{v, w\}$ is an edge in $G$. Therefore, Condition (ii) also holds. ■

**High-Level Description:**

1. Construct graph $G'(V - L, E')$ by deleting from $G$ the nodes in $L$ and the edges incident on those nodes.
2. **if** ($G'$ is not connected) **then output** "There is no solution" and **stop**.
3. Construct a minimum spanning tree $T'$ for $G'$.
4. **for** each node $v \in L$ **do** // Attach node $v$ to $T'$ using an edge of minimum cost.
   (a) Find the subset $S_v$ of $V - L$ such that $v$ is adjacent to each node $w \in S_v$ in $G$.
   (b) **if** ($S_v$ is empty) **then output** "There is no solution" and **stop**.
   (c) Find a node $w \in S_v$ such that the weight of the edge $\{v, w\}$ is a minimum over all the edges between $v$ and the nodes in $S_v$.
   (d) Attach $v$ as a child of $w$ in $T'$.
5. Output the spanning tree $T$ that results at the end of Step 4.

**Proof of correctness:** The fact that the above algorithm outputs a spanning tree $T$ where every node of $L$ is a leaf if and only if such a tree exists is a direct consequence of Lemma 1. We now show that among all the spanning trees of $G$ in which the nodes in $L$ appear as leaves, $T$ has the smallest cost.

Let $T^*$ be an optimal spanning tree of $G$ in which all the nodes in $L$ appear as leaves. We can divide the cost of $T^*$ into two parts: (i) the cost $C_1(T^*)$ of the spanning tree $T^*_{G'}$ for $G'(V-L, E')$ and (ii) the total cost $C_2(T^*)$ of the edges that attach each node of $L$ to the spanning tree $T^*_{G'}$. Thus, $C(T^*) = C_1(T^*) + C_2(T^*)$. Define $C_1(T)$ and $C_2(T)$ for the spanning tree $T$ for $G'(V-L, E')$ constructed by the algorithm in an analogous manner. Note that the cost of $T$, denoted by $C(T)$, is given by $C(T) = C_1(T) + C_2(T)$. Since the algorithm constructs a minimum spanning tree of $G'$, we have $C_1(T) \leq C_1(T^*)$. Further, since Step 4(c) of the algorithm attaches each node $v \in L$ to a node $w$ of the spanning tree of $G'$ such that the cost of the edge $\{v, w\}$ is a minimum over all the edges between $v$ and the nodes in $V - L$, it follows that $C_2(T) \leq C_2(T^*)$. As a consequence, $C(T) = C_1(T) + C_2(T)$ $\leq C_1(T^*) + C_2(T^*) = C(T^*)$. Thus, the tree $T$ produced by the algorithm is optimal. ∎

**Running Time Analysis:** We will consider each step of the algorithm separately. We use the standard notation that $n = |V|$ and $m = |E|$.

**Step 1:** We can implement this step in time $O(m \log n)$ as follows. We first sort the nodes in $L$. Since $|L| \leq n$, this can be done in $O(n \log n)$ time. To construct the adjacency list for $G'$ from that of $G$, we proceed as follows. For each node $v \in V$, if $v \in L$ (which can be determined using binary search in $O(\log n)$ time), we delete the adjacency list of $v$; otherwise, we go through the adjacency list of $v$ and delete each node $x$ such that $x$ appears in $L$. Thus, for each node $v$, this can be done in time $O(\text{degree}(v) \log n)$. So, the total time for Step 1 is $O(\sum_{v \in V} \text{degree}(v) \log n) = O(m \log n)$.

**Step 2:** Checking whether $G'$ is connected can be done in $O(m + n)$ time using breadth-first or depth-first search.

**Step 3:** Constructing a minimum spanning tree for $G'$ can be done using Kruskal's or Prim's algorithm in $O(m \log n)$ time.

**Step 4:** To implement this step, we first sort the nodes in $V - L$. Since $|V - L| \leq n$, the sorting step can be done in $O(n \log n)$ time. Now, for each node $v \in L$, we can go through the adjacency list of $v$, construct the set $S_v$ (i.e., the set of nodes in $V - L$ to which $v$ is adjacent in $G$) using a binary search of $V - L$. During this process, we can also find a node $w$ such that the edge $\{v, w\}$ has the minimum cost (among all the edges between $v$ and the nodes in $S_v$) in time $O(\text{degree}(v) \log n)$. So, the total time for Step 4 is $O(\sum_{v \in L} \text{degree}(v) \log n) = O(m \log n)$ (since $L \subseteq V$).

**Step 5:** Since $T$ has $n$ nodes and $n - 1$ edges, Step 5 can be done in $O(n)$ time.

By considering the times for Steps 1 through 5, it can be seen that the overall running time of the algorithm is $O(m \log n)$.

---

<u>**Problem 2:**</u>

**Terminology:** In the implementation of Prim's Algorithm discussed below, we use two queues, each implemented as a doubly-linked list. We use the term 'node' to refer to an item in these lists. We use the term 'vertex' to refer to a node of the given graph $G(V, E)$. As usual, $n = |V|$ and $m = |E|$.

**Idea:** Since there are only two possible edge weights (namely, 0 and 1), we don't need a general priority queue which may use $O(\log n)$ time for each Extract_Min and Decrease_Key operation.

Instead, we use two queues, denoted by $Q_0$ and $Q_1$, with $Q_0$ ($Q_1$) containing vertices which have not been added to the tree and whose current key values are 0 (1). If $Q_0$ is non-empty, we remove a node from $Q_0$ and add the corresponding vertex to the current tree; otherwise, we remove a node from $Q_1$ and add the corresponding vertex to the current tree. As noted below, each of these queue operations can be implemented to run in $O(1)$ time, and this leads to the desired running time of $O(m + n)$.

**(a) Data structures used:**

- Two doubly-linked lists $Q_0$ and $Q_1$. Each node in these lists has three fields: `vertex-id` (an integer, indicating the corresponding vertex of the graph), `previous` and `next` (two pointers, which point to the next and previous nodes of the corresponding list). Each node in $Q_0$ ($Q_1$) represents a vertex whose current key value is 0 (1) and which has not yet been added to the MST.

  **Note:** We use doubly-linked lists for the following reason: given a pointer to a node in a doubly-linked list, the node can be deleted from the list in $O(1)$ time. Also, a node can be inserted into a doubly-linked list in $O(1)$ time.

- An array `vertex-info` of size $n$. Element `vertex-info`$[i]$ of this array is a record containing the following information about vertex $v_i$ of $G$ ($1 \leq i \leq n$):

  (i) `key`: The current key value of the vertex. (This value can only be 0, 1 or $\infty$.)

  (ii) `parent`: A node $u$ such that $w(u, v_i)$ is the key value of $v_i$.

  (iii) `in-tree`: A Boolean value indicating whether or not the vertex has been added to the MST.

  (iv) `pointer`: The pointer to the node with `vertex-id` equal to $i$; This node may be in $Q_0$ or $Q_1$.

  **Note:** To keep the pseudocode description simple, we use `key(v)` to refer to the `key` value stored in the record for vertex `v`. (A similar notation is used for the other fields of the record.)

- For each node $u$, its adjacency list is denoted by Adj$[u]$ (as usual).

**(b) Modified Pseudocode for Prim's Algorithm:**

**Note:** Let `r` be the (given) root vertex for the MST to be constructed. The variable `mst-size` represents the number of vertices in the current tree.

```
1. for each vertex v do {
      key(v) = infinity;  parent(v) = NULL;  in-tree(v) = false;  pointer(v) = NULL;
   }

2. Set key(r) = 0 and mst-size = 1.

3. (a) Create a node (of a doubly-linked list) with vertex-id set to r and
       next and previous pointers set to NULL. Insert this node into list Q0.

   (b) Initialize Q1 to empty.
```

```
4. while (mst-size < n) do {

    (A) if (Q0 is not empty)
            Remove the first node from Q0.
        else
            Remove the first node from Q1.

    (B) Let u be the vertex-id stored in the node just removed.
        Set in-tree(u) = true;  mst-size++;

    (C) for each vertex v in Adj[u] do {
             if (in-tree(v) = false) { // No need to consider nodes already in the tree.
                 if (key(v) = infinity) { // key(v) will change to 0 or 1.
                    (i) key(v) = w(u,v);  parent(v) = u;
                   (ii) Create a node (of a doubly-linked list) with
                        vertex-id set to v and previous and next pointers
                        set to NULL. Let pointer(v) point to this node.
                  (iii) if (key(v) = 0)
                           Insert the node created in Step 4(C)(ii) into Q0.
                        else
                           Insert the node created in Step 4(C)(ii) into Q1.
                 } // End of key(v) = infinity case.
                 else { // key(v) is 0 or 1.
                   if (key(v) = 1) { // Key value may change to 0.
                      if (w(u,v) = 0) {
                         (i) key(v) = 0;  parent(v) = u;
                        (ii) Use pointer(v) to remove the node with
                                vertex-id = v from Q1 and insert it into Q0.
                      }
                   }
                 }
             } // End of outermost if.
        } // End of for loop (of Step 4(C)).
   } // End of while loop (Step 4).
```

**(c) Running time analysis:** Step 1 runs in $O(n)$ time. Steps 2 and 3 run in $O(1)$ time. We analyze Step 4 as follows.

- The **while** loop of Step 4 runs at most $n$ times. For each iteration of this loop, Steps 4(A) and 4(B) run in $O(1)$ time. So, over all the $n$ iterations, Steps 4(A) and 4(B) use $O(n)$ time.
- For each vertex $u$, Step 4(C) examines the adjacency list of $u$. For each node $v$ in Adj$[u]$, the time used in Step 4(C) is $O(1)$ since we can remove and insert a node from a doubly-linked list in $O(1)$ time. So, the time spent in Step 4(C) for each vertex $u$ is $O(\text{degree}(u))$. Thus, over all the iterations of the **while** loop of Step 4, the total time spent in Step 4(C) is $O(\sum_{u \in V} \text{degree}(u)) = O(m)$, since $\sum_{u \in V} \text{degree}(u) = 2m$.

Thus the total time for Step 4 is $O(m+n)$. Since this dominates the running time, the overall running time of the algorithm is also $O(m+n)$.

---

**Problem 3:**

As defined in the problem, a **special** node in a directed graph $D(V, A)$ is one whose indegree $= n-1$ and whose outdegree $= 0$. (Recall that $n = |V|$.) We first show that each directed graph (without self loops) has *at most one* special node.

**Lemma 2:** Each directed graph $D(V, A)$ (without self loops) has *at most one* special node.

**Proof:** The proof is by contradiction. Suppose $D(V, A)$ has two distinct special nodes $v$ and $w$. If there is an edge from $v$ to $w$, then the outdegree of $v$ is 1 and so $v$ cannot be a special node. On the other hand, if there is no edge from $v$ to $w$, then the indegree of $w$ is *less than* $n-1$, and so $w$ cannot be a special node. We get a contradiction in each case and the lemma follows. ■

**Idea:** Initially, all nodes in $V$ are candidates for the special node. Let $M$ denote the adjacency matrix of $G$. Consider any pair of nodes $v_i$ and $v_j$ and suppose we check the entry $M[i, j]$.

(a) If $M[i, j] = 0$, then there is no edge from $v_i$ to $v_j$; thus, the indegree of $v_j$ will be *less than* $n-1$. Therefore, $v_j$ *cannot* be the special node.

(b) If $M[i, j] = 1$, then there is an edge from $v_i$ to $v_j$; thus, the outdegree of $v_i$ cannot be 0. Therefore, $v_i$ *cannot* be the special node.

Hence, checking one entry of $M$ allows us to *eliminate* one node from the set of candidates for the special node. So, after $n-1$ checks of $M$, we can reduce the number of candidates to 1. If the remaining candidate is node $v_i$, then we can compute the indegree and outdegree of $v_i$ (by probing all the entries of row $i$ and column $i$ of $M$) and determine whether or not $v_i$ is special. We need to check at most $2n$ entries of $M$ to compute indegree($v_i$) and outdegree($v_i$). Thus, overall, the algorithm examines only $O(n)$ entries of $M$.

**High-Level Description:**

1. Let Current = 1.
2. **for** $i = 2$ **to** $n$ **do** // Eliminates nodes until only one candidate remains.
   **if** ($M$[Current, $i$] = 1) **then** Current = $i$.
3. Compute the indegree $\alpha$ and the outdegree $\beta$ of the node given by Current.
4. **if** ($\alpha = n-1$ **and** $\beta = 0$)
   **then** Output Current as the special node
   **else** Output "No special node".

**Correctness of the Algorithm:** This is a direct consequence of Lemma 2 and the discussion presented under "Idea".

**Number of Probes Used:** Step 2 of the algorithm uses $n-1$ probes of the matrix $M$. Step 3 probes at most $2n$ entries of $M$. Hence, the total number of probes of $M$ is at most $3n-1 = O(n)$.

---