CPS 130 Homework 19 - Solutions

1. [CLRS 22.1-1]

   Solution: Given an adjacency-list representation *Adj* of a directed graph, the out-degree of a vertex u is equal to the length of $Adj[u]$, and the sum of the lengths of all the adjacency lists in *Adj* is $|E|$. Thus the time to compute the out-degree of every vertex is $\Theta(V+E)$. The in-degree of a vertex u is equal to the number of times it appears in all the lists in *Adj*. If we search all the lists for each vertex, the time to compute the in-degree of every vertex is $\Theta(VE)$. Alternatively, we can allocate an array T of size $|V|$ and initialize its entries to zero. Then we only need to scan the lists in *Adj* once, incrementing $T[u]$ when we see u in the lists. The values in T will be the in-degrees of every vertex. This can be done in $\Theta(V+E)$ time with $\Theta(V)$ additional storage.

   The adjacency-matrix A of any graph has $\Theta(V^2)$ entries, regardless of the number of edges in the graph. For a directed graph, computing the out-degree of a vertex u is equivalent to scanning the row corresponding to u in A and summing the ones, so that computing the out-degree of every vertex is equivalent to scanning all entries of A. Thus the time required is $\Theta(V^2)$. Similarly, computing the in-degree of a vertex u is equivalent to scanning the column corresponding to u in A and summing the ones, thus the time required is also $\Theta(V^2)$.

2. [CLRS 22.1-5]

   Solution: To compute G^2 from the adjacency-list representation *Adj* of G, we perform the following for each $Adj[u]$:

   ```
   for each vertex v in Adj[u]
       for each vertex w in Adj[v]
           edge(u,w) ∈ E^2
           insert w in Adj2(u)
   ```

   where *Adj*2 is the adjacency-list representation of G^2. After we have computed *Adj*2, we have to remove any duplicate edges from the lists (there may be more than one two-edge path in G between any two vertices). For every edge in *Adj* we scan at most $|V|$ vertices, we compute *Adj*2 in time $O(VE)$. Removing duplicate edges is done in $O(V+E)$ as shown in [CLRS 22.1-4]. Thus the total running time is $O(VE)+O(V+E)= O(VE)$.

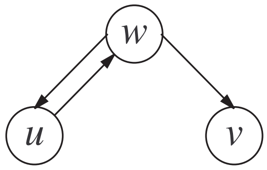
   Let A denote the adjacency-matrix representation of G. The adjacency-matrix representation of G^2 is the square of A. Computing A^2 can be done in time $O(V^3)$ (and even faster, theoretically; Strassen's algorithm for example will compute A^2 in $O(V^{\lg 7})$).

3. [CLRS 22.2-3]

   Solution: If the input graph for BFS is represented by an adjacency-matrix A and the BFS algorithm is modified to handle this form of input, the the running time will be the size of A, which is $\Theta(V^2)$. This is because we have to modify BFS to look at every entry in A in the `for` loop of the algorithm, which may or may not be an edge.

4. [CLRS 22.3-7]

   Solution: Consider the following directed graph G:



   There is a path from u to v in G. Suppose a DFS search discovers vertices in the order w, u, v. Then the depth-first tree will have root w and u, v are children of w. However, v is not a descendant of u.

   This is just one possible counterexample.

5. [CLRS 22.4-5]

   Solution: We can perform topological sorting on a directed acyclic graph G using the following idea: repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. To implement this idea, we first create an array T of size $|V|$ and initialize its entries to zero, and create an initially empty stack S. Let Adj denote the adjacency-list representation of G. We scan through all the edges in Adj, incrementing $T[u]$ each time we see a vertex u. In a directed acyclic graph there must be at least one vertex of in-degree 0, so we know that there is at least one entry of T that is zero. We scan through T a second time and for every vertex u such that $T[u] = 0$, we push u on S. Pop S and output u. When we output a vertex we do as follows: for each vertex v in $Adj[u]$ we decrement $T[v]$ by one. If any of these $T[v] = 0$, then push v on S.

   To show our algorithm is correct: At each step there must be at least one vertex with in-degree 0, so the stack is never empty, and every vertex will be pushed and popped from the stack once, so we will output all the vertices. For a vertex v with in-degree $k \geq 1$, there are k vertices $u_1, u_2, \ldots u_k$ which will appear before v in the linear ordering of G. Then $T[v] = k$, since $v \in Adj[u_i]$ for $i = 1, \ldots, k$ vertices of G, and v will only be pushed on the stack after all u_i have already been popped (each pop decrements $T[v]$ by one).

   The running time is $\Theta(V)$ to initialize T, $O(1)$ to initialize S, and $\Theta(E)$ to scan the edges of E and count in-degrees. The second scan of T is $\Theta(V)$. Every vertex will be pushed and popped from the stack exactly once. The $|E|$ edges are removed from the graph once (which corresponds to decrementing entries of T $\Theta(E)$ times). This gives a total running time of $\Theta(V)+O(1)+\Theta(E)+\Theta(V)+\Theta(E) = \Theta(V + E)$.

   If the graph has cycles, then at some point there will be no zero entries in T, the stack will be empty, and our algorithm cannot complete the sort.

   Note: The algorithm to solve this problem is also given in Lecture 19, but you still need to analyze the running time and prove it works.