CSci 231 Homework 10 Solutions *

Basic Graph Algorithms

1. [CLRS 22.1-1] Describe how to compute the in-degree and out-degree of the vertices of a graph given its (1) adjacency -list representation and (b) adjacency-matrix representation.

   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 one vertex is $\Theta(|Adj(v)|)$ and for all vertices 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 all vertices 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 $\Theta(V)$ for one vertex, and $\Theta(V^2)$ for all vertices. 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 $\Theta(V)$ for one vertex, and $\Theta(V^2)$ for all vertices.

2. [CLRS 22.1-5] Give and analyse an algorithm for computing the square of a directed graph G given in (a) adjacency-list representation and (b) adjacency-matrix representation.

   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)
```

*Collaboration is allowed and encouraged, if it is constructive and helps you study better. Remember, exams will be individual. Write the solutions individually, and list the names of the collaborators along with the solutions.

where $Adj2$ is the adjacency-list representation of G^2. For every edge in Adj we scan at most $|V|$ vertices, thus we compute $Adj2$ in time $O(VE)$.

After we have computed $Adj2$, 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). Removing duplicate edges is done in $O(V+E')$ where $E' = O(VE)$ is the number of edges in $Adj2$ (see for instance problem 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 22.1-7) The *incidence matrix* of a directed graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = [b_{ij}]$ such that

$$b_{ij} = \begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i \\ 1 & \text{if edge } j \text{ enters vertex } i \\ 0 & \text{otherwise} \end{cases}$$

Describe what the entries of the matrix product $B \times B^T$ represent, where B^T is the transpose of B.

Solution:

$BB^T(i,j) = \sum_{e \in E} b_{ie} b_{ej}^T = \sum_{e \in E} b_{ie} b_{je}$

If $i = j$ then $b_{ie}b_{je} = 1$ (it is $1 \cdot 1$ or $-1 \cdot -1$) whenever i enters or leaves vertex i, and 0 otherwise.

If $i \neq j$, then $b_{ie}b_{je} = -1$ when $e = (i, j)$ or $e = (j, i)$ and 0 otherwise.

Thus

$$BB^T(i,j) = \begin{cases} indegree(i) + outdegree(i) & \text{if } i = j \\ -(\text{nb of edges connecting } i \text{ and } j) & \text{if } i \neq j \end{cases}$$

4. (CLRS 22.2-3) Analyse BFS running time if the graph is represented by an adjacency-matrix.

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 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.

5. (CLRS 22.2-8) Consider an undirected connected grah G. Give an $O(V+E)$ algorithm to compute a path that traverses each edge of G exactly once in each direction.

Solution: Perform a DFS of G starting at an arbitrary vertex. The path required by the problem can be obtained from the order in which DFS explores the edges in the graph. When exploring an edge (u, v) that goes to an unvisited node the edge (u, v)

is included for the first time in the path. When DFS backtracks to u again after v is made BLACK, the edge (u, v) is included for the 2nd time in the path, this time in the opposite direction (from v to u). When DFS explores an edge (u, v) that goes to a visited node (GRAY or BLACK) we add $(u, v)(v, u)$ to the path. In this way each edge is added to the path exactly twice.

6. (CLRS 22.4-3) Given an undirected graph $G = (V, E)$ determine in $O(V)$ time if it has a cycle.

   Solution: There are two cases:

   (a) $E < V$: Then the graph may or may not have cycles. To check do a graph traversal (BFS or DFS). If during the traveral you meet an edge (u, v) that leads to an already visited vertex (GRAY or BLACK) then you've gotten a cycle. Otherwise there is no cycle. This takes $O(V + E) = O(V)$ (since $E < V$).

   (b) $E \geq V$: In this case we will prove that the graph must have a cycle.

   Claim 1: A tree of n nodes has $n - 1$ edges.

   Proof of claim 1: By induction. Base case: a tree of 1 vertex has 0 edges. ok. Assume inductively that a tree of n vertices has $n - 1$ edges. Then a tree T of $n + 1$ vertices consists of a tree T' of n vertices plus another vertex connected to T' through an edge. Thus the number of edges in T is the number of edges in T' plus one. By induction hyopthesis T' has $n - 1$ edges so T has n edges. qed.

   Coming back to the problem: Assume first that the graph G is connected. Perform a DFS traversal of G starting at an arbitrary vertex. Since the graph is connected the resulting DFS-tree will contain all the vertices in the graph. By Claim 1 the DFS-tree of G has $V - 1$ edges. Therefore since $E \geq V$ there will be at least an edge in G which is not in the DFS-tree of G. This edge gives a cycle in G.

   If the graph G is not connected: If G has 2 connected components $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Then it is easy to prove, by contradiction, that $E \geq V$ implies that either $E_1 \geq V_1$ or $E_2 \geq V_2$ (or both). In either case either G_1 will have a cycle or G_2 will have a cycle (or both).

   (If the graph G is not connected and has k connected components then the same argument as above works, except that formally we need induction on k).

7. (CLRS 22.4-5) Give an algorithm to compute topological order of a DAG without using DFS.

   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.

8. (CLRS 22-4) Let $G = (V, E)$ be a directed grah in which each vertex $u \in V$ is labeled with a unique integer $L(u)$ from the set $\{1, 2, ..., |V|\}$. For each vertex $u \in V$, let $R(u) = \{v \in V | u \text{ reaches } v\}$ be the set of vertices that are reachable from u. Define $\min(u)$ to be the vertex in $R(u)$ whose label is minimum, i.e. $\min(u)$ is the vertex v such that $L(v) = \min\{L(w)|W \in R(u)\}$. Give an $O(V + E)$-time algorithm that computes $\min(u)$ for all vertices $u \in V$.

   Solution: One solution is to compute the strongly connected components of the graph and erase all but the smallest label vertex in each component C; let this vertex be denoted $w(C)$. For every edge (u, v) with u not in the C and v in C add an edge (u, w). For every edge (v, u) with v in C and u not in C add an edge (w, u).(this process is called *contracting* C to a single vertex w). The resulting graph is a DAG. This DAG can be computed in $O(V + E)$ time (since strongly connnected components can be computed in $O(V + E)$ time). So we reduced the problem to the same problem on a DAG. Now it is simple: traverse the graph in reverse topological order. Initially every vertex has $min(u) = u$. For every vertex u look at its outgoing edges (u, v) and update $min(u) = \min\{min(v)|(u, v)\}$. Since We traverse vertices in reverse topological order all outgoing vertices (u, v) of u will have already found their final label $min(v)$.

   A much simpler way to solve this problem (without worrying about strongly connected compoenents) is to traverse the graph (either BS or DFS) but looking at the *incoming* edges rather than at outgoing edges, while processing vertices in increasing order of their label. The formal way to say this is as follows: compute a reverted graph G^T

which is the same as G but with the direction of every edges reverted. This graph can be easily computed in linear time $O(V + E)$. Then

```
sort vertices in increasing order of their label
for each v in order do
    if v not black then BFS(v)
```

That is, first perform $BFS(1)$; this will visit all vertices reachable from 1 in G^T (that is, which can reach 1 in G) and set their $min(u) = 1$. Then find the next smallest node that has not been reached in the previous BFS and start BFS from it, and so on.

This in total takes $O(V)$ to sort the vertices (using a linear-time sorting algrithm) and $O(V + E)$ to do the graph traversal (BFS or DFS).