Stefanie Shidoosh

CS 180: HW #3

1. Given two nodes u and v in a directed rooted tree T, design an algorithm that finds the lowest common ancestor in $O(h)$ time, where h is the distance between u and v in the underlying undirected version of T.

    Preprocessing: Using Breadth-First Search, find the distances of all v in V(T) with respect to the root node.
        Visit root node
        Visit its children
        Once all children visited, recursively perform the aforementioned steps on each child (which then become the "root") in order of original visit, noting distance from original root via incrementing every time this step is performed.

    Identify the ancestors of the vertices u and v, respectively,
    Group the ancestors of u and v per distance from the root
    Search the aforementioned groups for a common ancestor with the greatest depth, resulting in the LCA

    Preprocessing with BFS takes $O(|E|)$ as each vertex is guaranteed at least one edge, and $|E| >= |V| - 1$.
    After this step, this algorithm takes $O(h)$ time, as in the worst case, identifying the ancestors of the given vertices would require traversing from a terminal node to the root. The proceeding steps are more trivial and are of lesser magnitude and thus do not contribute to the time complexity of the algorithm in a meaningful way.

Stefanie Shidoosh

2. In some country, there are n flights among n cities. Each city has an exactly one outgoing flight but may have no incoming flight or multiple incoming flights. Design an algorithm to find the largest subset S of cities such that each city in S has at least one incoming flight from another city in S.

   Note: Each v in V(S) must have an in-degree of at least 1.

   Let cities be v in V(G). Let incoming and outgoing flights be $e\ in\ E(G)$.
   Let $G = (V, E)$ be a directed graph.
   Let G be represented by an $n \times n$ adjacency matrix M, where n is the number of vertices, which is a (0,1)-matrix such that $M[i, j] = 1$ if and only if there is a directed edge from v_i to v_j.

   Thus, a city, v_i , has an incoming flight if there exists at least one 1 in the i^{th} column of M. With this information, the subset S can be derived:

   Initialize all cells in M = 0
   For each pair of arbitrary vertices v_i and v_j
   If the edge (v_i, v_j) exists, M[i, j] = 1
   As reference point, find the j^{th} column with the greatest amount of 1 entries. Use this vertex and its neighbors initially.
   For each column in M that correspond to v_j's neighbor
   If column has at least one 1 entry, allocate the corresponding vertices to the subset S

   This implementation derives the in-degree data structure (the columns with at least one 1 entry) which is then used to aggregate the largest subset of edges that are incident on the vertices in $V(S)$. The time complexity of this algorithm is $O(N)$ as in the worst case, the subset S would be all v in $V(G)$, which is n.

Stefanie Shidoosh

3. A directed graph $G = (V, E)$ is acyclic if there is no cycle in G. Given a directed graph G, design an $O(|E|)$ algorithm to determine whether it is acyclic or not.

    To determine if a directed graph G is acyclic (DAG):
        If it can be topologically sorted, it is acyclic
        Else, it must have a cycle.

    Considering this, we can modify the topological sorting algorithm for our purposes:
        Store the in-degree of each v in V(G)
        Initialize a queue with in-degree 0 vertices
        While there are vertices in the queue:
            Dequeue an arbitrary vertex (Note: can output the vertex if interested in the actual sorting)
            Reduce in-degree of all vertices adjacent to it by 1
            Remove the edge of the graph
            Enqueue any of the vertices whose in-degree became 0
        If the graph has remaining edges, it has a cycle
        Else, it is acyclic.

    The time complexity of topological sorting is $O(|V| + |E|)$, however, we may assume that every vertex as at least one edge. Therefore, $|E| >= |V| - 1$, thus $|E| + |V| = O(|E| + |E| + 1) = O(|E|)$ follows. Thus, for such a graph, $O(|E|)$ *and* $O(|V| + |E|)$ are equivalent complexities.

Stefanie Shidoosh

4. Given an arbitrary DAG $G = (V, E)$, give an $O(|E|)$ algorithm to solve topological sorting on G by using DFS. Argue the correctness of your algorithm and its $O(|E|)$ time complexity.

   Note: Provide a topological sorting algorithm for any DAG by using DFS and show the proof of the correctness (Piazza).

   Proof of Correctness

   There are 4 different types of edges considered in DFS for a directed graph: Tree edge, back edge, forward edge, and cross edge.
   The tree, forward, and cross edges all maintain the properties such that each edge (u, v) of these categories, t[u] > t[v], where t is finish time.
   Back edges do not occur in DAGs by definition, so the algorithm will not concern itself with this type, as if one existed it would make it prone to cyclical nature.
   Therefore, by ordering the vertices in decreasing order with respect to their finish time, the result is a topological sorting of the graph.
   This works with any DAG as all DAGs must have these properties.

   Algorithm

   ```
   For all v in V(G)
       Run DFS
           Mark v as visited
           Visit v's neighbors
               If not visited, mark as visited
           Note the finish time, save the vertex into a data
           structure, with the index corresponding to the time,
           which increments at each discovered vertex
       The reverse order of the data structure will then be sorted
       topologically.
   ```

   The time complexity for DFS is $O(|V| + |E|)$ usually, but we can assume that all v in V(G) has at least one edge (Piazza), and thus we can assume that the traversal of G is bounded by the vertices ($|E| >= |V| - 1$).
   Therefore, this algorithm is $O(|E|)$, due to this assumption as the set of vertices V is essentially implied in the set of edges, E.