Solutions to Quiz 1

Solution 1:

(a) (iv) An undirected graph has at most $\binom{n}{2}$ edges, consisting of all unordered pairs of size two.

(b) (ii) A directed graph has at most n^2 vertices, consisting of all ordered pairs of size two, including self loops.

(c) (iv) A DAG has at most $\binom{n}{2}$ edges, which consists of the edges (i, j), where $1 \leq i < j \leq n$.

Solution 2: Here are the asymptotic relationships. Justifications (which were not required) are given below.

$$\begin{array}{llcccc} \text{(a)} & n \log n & \prec & n\sqrt{n} & \prec & n^2 \\ \text{(b)} & 2^{\lg n} & \approx & n^{\lg 2} & \prec & 2^{(2 \lg n)} \\ \text{(c)} & n \log(\log n) & \prec & n \log n & \prec & n(\log n)^2 \end{array}$$

(a) Ignoring the common factor of n, these follow because $\log n \prec n^{1/2} \prec n$.

(b) Follows from the facts that $2^{\lg n} = n$, $n^{\lg 2} = n$, and $2^{2 \lg n} = (2^{\lg n})^2 = n^2$.

(c) We can ignore the common factor of n. Since $\log m \prec m$ and setting $m = \log n$, it follows that $\log \log n \prec \log n$. Also, since $1 \prec \log n$, by multiplying by $\log n$, we have $\log n \prec (\log n)^2$.

Solution 3: (i) It is best to process vertices/edges in (forward) topological order. (Note that other visit orders will still produce a correct result, but the running time will be larger.)

A general principle of the Bellman-Ford algorithm is that when relaxations are performed in order along the shortest path, then the distance values along the path are correct. This is because we learn the distance value to each vertex before we propagate it to the next. The good thing about a DAG is that all paths (not just shortest paths) respect the topological order, and hence a single pass of Bellman-Ford in forward topological order will fix all the distances to their correct value. This implies that the running time is $O(n + m)$. The other orders may result in as many as $n - 1$ passes, for a total running time of $O(nm)$.

Solution 4:

(a) See Fig. 1. Each vertex is labeled with its discovery and finish times ($d[u]/f[u]$). Tree edges are solid and the other edges are dashed and labeled by their type.

(b) The strong components are $\{a, c, d, e, f\}$, $\{b\}$, and $\{g\}$.

Solution 5: As with the alternating paths problem from the practice set, we need to keep separate counters based on the type of paths. In our case, there two types of paths that are relevant: 0-paths that consist of nothing by 0-label edges, and 1-paths, those that consist of a single 1-edge. For each $u \in V$, let $C_0[u]$ denote the numbers of 0-paths originating at u, and let $C_1[u]$ denote the number of 1-paths originating at u. We compute these values using DFS. As we are visiting a vertex u, for each edge (u, v) we have two cases:

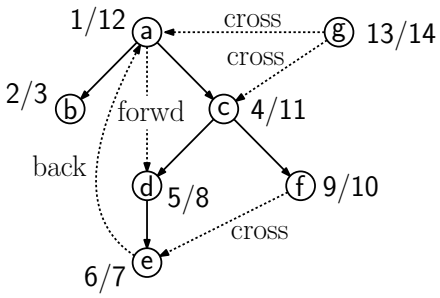

Figure 1: Problem 3: Depth First Search.

- If label$(u, v) = 0$, we can form 0-paths by combining this edge with any 0-path out of v. We can take the single edge (u, v). Thus, $C_0[u]$ += $1 + C_0[v]$.

  We can form 1-paths by combining this edge with all 1-paths out of v, which implies that $C_1[u]$ += $C_1[v]$.

- If label$(u, v) = 1$, we cannot use the edge to form any new 0-paths. We can form 1-paths either by taking this edge alone, or by combining this edge with all the 0-paths out of v. This implies that $C_1[u]$ += $1 + C_0[v]$.

This can be computed efficiently by DFS as shown in the following code block.

```
onePaths(G=(V,E))                       // count paths with a single 1-edge
    for each (u in V) mark[u] = undiscovered    // initialize
    for each (u in V)                   // visit all vertices
        if (mark[u] == undiscovered) countPaths(u)
    for each (u in V) one[u] = C1[u]    // final counts
    return one

countPaths(u)
    mark[u] = visited
    C0[u] = C1[u] = 0                   // initialize counts
    for each (v in Adj[u])
        if (mark[v] == undiscovered) altCountPaths(v)  // count v's paths
        switch (label(u,v))
            0: C0[u] += 1 + C0[v]       // accumulate 0-paths
               C1[u] += C1[v]           // include v's 1-paths
            1: C1[u] += 1 + C0[v]       // include (u,v) and v's 0-paths
    mark[u] = finished
```

An example is shown in Fig. 2. Correctness follows from the above derivation. A key element is the fact that, since G is a DAG, we access $C_0[v]$ and $C_1[v]$ only after calling `countPaths(v)`, which implies that these values are correctly computed. Since G is a DAG, whenever we access the count values of any vertex, this vertex has already "finished," and hence these counts are correct (by a simple induction argument). As with any DFS algorithm, the running time is $O(n + m)$.

You might wonder whether it is possible to count 1-paths one-by-one. This cannot work efficiently, since a DAG may have exponentially many paths. Consider the DAG in Fig. 3, which consists of ℓ copies of the same structure. Let's just consider 1-paths from each node that go all the

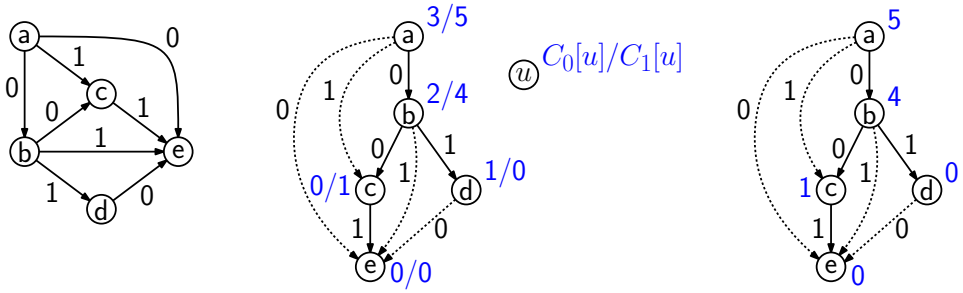

Figure 2: Counting 1-paths with $C_0[u]$ and $C_1[u]$ shown for each vertex in blue.

way to the final rightmost node. A path with a single 1-edge can be formed by selecting a single 1-edge from any one of the levels ℓ, and for all the remaining $\ell - 1$ levels, it can either take the upper track or lower track, for a total of $2^{\ell-1}$ possible paths. Thus, the total number of 1-paths from each node to the end is $\ell \cdot 2^{\ell-1}$. Since $n = 3\ell + 1$, this is exponential in n.

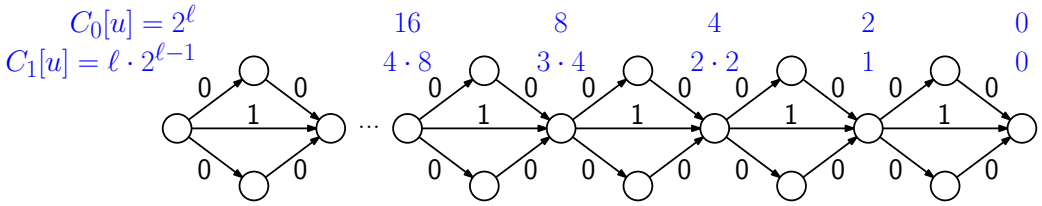

Figure 3: There are exponentially many 1-paths (even restricted to maximal paths).

The partial credit part (find all paths with at least one 1-edge) is similar but a bit simpler. As before, we maintain two counts, $C[u]$ stores *all* the paths starting at u, and $C_1[u]$ stores all the paths that have at least one 1-edge. To count all paths, for each edge (u, v) (irrespective of label) we count this edge and all paths out of v, implying that $C[u]$ += $1 + C[v]$. To compute C_1, if this is a 0-edge, we just take $C_1[v]$, implying that $C_1[u]$ += $C_1[v]$. If this is a 1-edge, we include (u, v) and all the paths out of v, implying that $C_1[u]$ += $1 + C[v]$.

```
altCountPaths(u)                          // count paths with at least 1-edge
    mark[u] = visited
    C[u] = C1[u] = 0                      // initialize counts
    for each (v in Adj[u])
        if (mark[v] == undiscovered) altCountPaths(v)  // count v's paths
        C[u] += 1 + C[v]                  // update count of all paths
        switch (label(u,v))
            0: C1[u] += C1[v]             // 0-edge: include v's 1-paths
            1: C1[u] += 1 + C[v]          // 1-edge: include (u,v) and all of v's paths
    mark[u] = finished
```