Solutions to Practice Problems 2

Solution 1:

We have omitted the drawing of G^R. For parts (b)–(d), see Fig. 1. The final strong components are $\{f, g\}$, $\{i, j, h\}$, $\{c, d, e\}$, and $\{a, b\}$.

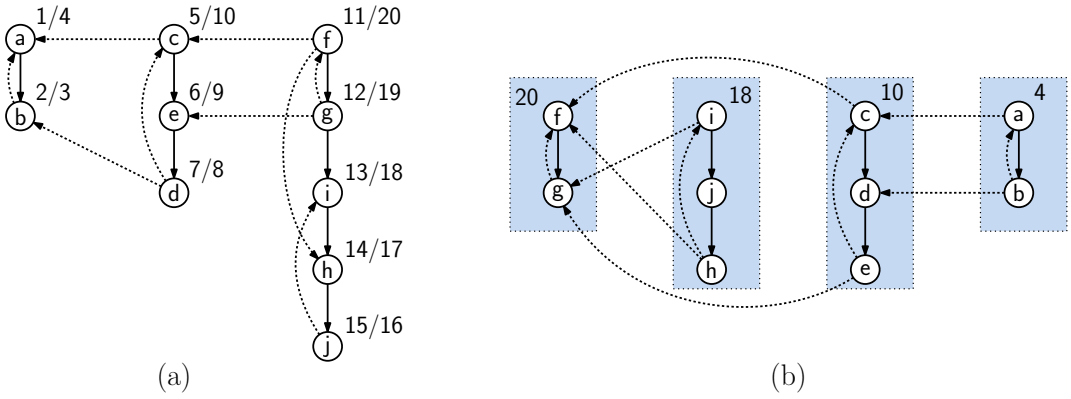

Figure 1: Strong components: (a) DFS on G^R and (b) final DFS with finish times shown for roots.

Solution 2: We use DFS. For each $u \in V$, let $A_0[u]$ and $A_1[u]$ denote the numbers of alternating paths originating at u starting with edges of labels 0 and 1, respectively. On first visiting a vertex, we initialize $A_0[u] \leftarrow A_1[u] \leftarrow 0$. Then for each neighbor v of u, if v has not yet been processed (`mark[v] = undiscovered`), we invoke DFS on v. Because G is a DAG, we know that v is "finished" after this, and therefore $A_0[v]$ and $A_1[v]$ have their correct and final values.

If label$(u, v) = 0$, we can form a 0-alternating path in two ways. First, we just use the single edge (u, v), and second, we prepend (u, v) onto all the 1-alternating paths starting at v. There are $A_1[v]$ such paths, which implies that

$$A_0[u] \mathrel{+}= 1 + A_1[v], \qquad \text{if label}(u, v) = 1.$$

Symmetrically, we have

$$A_1[u] \mathrel{+}= 1 + A_0[v], \qquad \text{if label}(u, v) = 0.$$

An example showing the values $A_0[u]$ and $A_1[u]$ is shown in Fig. 2. For example, we have

$$A_1[\mathsf{b}] \;\leftarrow\; (1 + A_0[\mathsf{e}]) + (1 + A_0[\mathsf{d}]) \;=\; (1 + 0) + (1 + 1) \;=\; 3.$$

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

Correctness follows from the above remarks, and the fact that we access $A1[v]$ and $A0[v]$ only after calling `countPaths(v)`, which implies that these values are correctly computed. Here we are making critical use of the fact that G is a DAG, since this implies that when 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 $O(n + m)$.

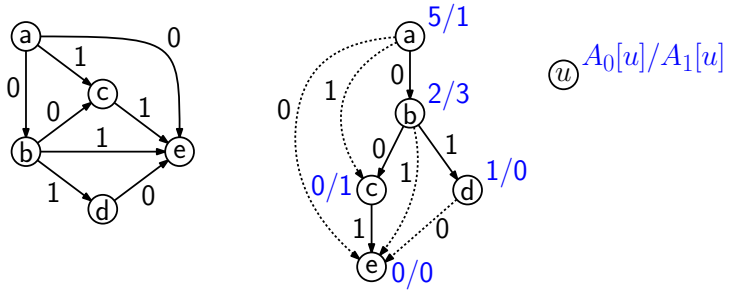

Figure 2: Counting alternating paths with $A_0[u]$ and $A_1[u]$ shown for each vertex in blue.

Counting Alternating Paths

```
altPaths(G=(V,E))
    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) alt[u] = A0[u] + A1[u]    // compute totals
    return alt

countPaths(u)
    mark[u] = visited
    A0[u] = A1[u] = 0                           // initialize counts
    for each (v in Adj[u])
        if (mark[v] == undiscovered) countPaths(v)  // count v's paths
        switch (label(u,v))
            0: A0[u] += 1 + A1[v]               // accumulate 0-paths
            1: A1[u] += 1 + A0[v]               // accumulate 1-paths
    mark[u] = finished
```

Solution 3:

(a) Recall from class that a graph has a cycle if and only if its DFS tree has a back edge. Since any spanning tree for a graph with n vertices has $n-1$ edges, it follows that if G has n edges or more, its DFS tree *must* have at least one back edge. As soon as the DFS has processed n edges, we must have seen a back edge, and hence we can stop the DFS.

It is easy to modify the DFS algorithm given in class to determine the existence of a back edge. In particular, when processing the neighbors v of a vertex u, we first check whether v is yet to be discovered. If so, we visit it. If not, then v must be an ancestor of u (because in the DFS of an undirected graph, there are only tree edges and back edges). We need to be a bit careful, since v may be u's parent in the tree (that is, we are just seeing the tree edge from the opposite end). To determine this, we check whether $\texttt{pred}[u] \neq v$. If so, v is a non-parent ancestor, implying that (u, v) is indeed a back edge, and we have found a cycle. We can determine the vertices on the cycle by tracing back the predecessor links from u until reaching v. The DFS visit code is presented in the following code block.

By the remarks made above, if we fail to discover such a vertex u, then G has at most $n-1$ edges, and hence DFS terminates in time $O(n+m) = O(n+(n-1)) = O(n)$, as desired. On

```
DFSvisit(u)                                  // perform a DFS search at u
    mark[u] = discovered                     // u has been discovered
    for each (v in Adj(u))
        if (mark[v] == undiscovered)         // undiscovered neighbor?
            pred[v] = u
            DFSvisit(v)                      // ...visit it
        else if (v != pred[u])               // (u,v) is a back edge
            cycleList = [u]                  // list the vertices in the cycle
            w = u
            do                               // trace back pred links to v
                w = pred[w]
                append w to cycleList
            while (w != v)
            terminate DFS and return cycleList
```

the other hand, as soon as we process the nth edge, this condition will be satisfied. Tracing the predecessor links takes $O(n)$ time. Therefore, the overall running time is $O(n)$.

(b) We claim that this will not work for directed graphs. Consider any algorithm that determines whether a digraph has a cycle. Construct a digraph $G = (V, E)$ whose vertex set is $V = \{1, \ldots, n\}$, and whose edge set consists of

$$E = \{(i, j) : 1 \le i < j \le n\}$$

(see Fig. 3(a)). Clearly, G is a DAG and has $\binom{n}{2} = \Omega(n^2)$ edges.

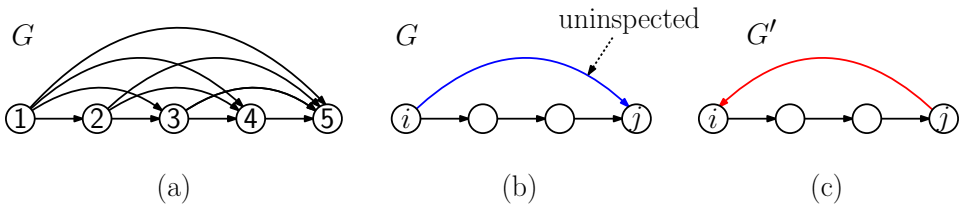

Figure 3: The DAG G.

Suppose to the contrary that there did exist an algorithm that could correctly determine whether a digraph has a cycle in $O(n)$ time. There must exist a constant c such that this algorithm inspects at most cn edges. No matter how large c is, we can find a sufficiently large value of n such that $\binom{n}{2} > cn$, which implies that this algorithm must fail to look at some edge of G.

Let's make the gap even bigger, by assuming that n is large enough that $\binom{n}{2} - 2n > cn$. This means that there are at least $2n$ edges that the algorithm has failed to inspect. Note that we can modify these uninspected edges however we like, and the algorithm cannot change its answer, since it never looked at any of them.

When presented with G, the hypothesized algorithm either answers "yes" or "no". If it answers "yes", then it is clearly wrong. On the other hand, if answers "no", we find a pair

(i, j), where $i < j - 1$, and the algorithm failed to inspect either (i, j) or (j, i) (see Fig. 3(b)). We know that such an edge must exist, because the only edges of G we have excluded are those where $i = j - 1$ and their reversals, of which there are at most $2n$.

Let us define a new graph G', which is identical to G, but we replace edge (i, j) with (j, i) (see Fig. 3(c)). As observed earlier, algorithm must yield the same output, since it looked at neither pair (i, j) nor (j, i). However, G' has a cycle $\langle j, i, i + 1, \ldots, j\rangle$, contradicting the algorithm's correctness.

Solution 4: The algorithm is quite simple, but it relies on an observation that is not entirely obvious. This is given in the following lemma.

Lemma: A DAG $G = (V, E)$ is semi-connected if and only if, for any topological ordering of the vertex set $V = \{1, \ldots, n\}$, $(i, i + 1) \in E$ for all i, where $1 \leq i < n$.

Proof: First observe that if this is true, then G has only one possible topological ordering. G is clearly semi-connected, because there is a path in G between any two vertices i and j, where $i < j$, and there is a path in the reversal of G if $j < i$ (see Fig. 4(a)).

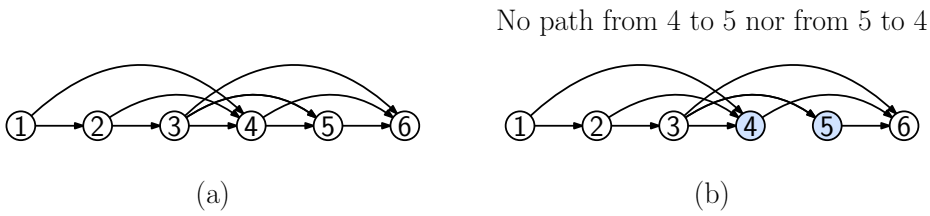

Figure 4: Topological ordering in a semi-connected DAG.

To show the converse, consider any topological ordering of the vertices of G. If, to the contrary, there was any pair $(i, i + 1)$, where this edge is not present in G (see Fig. 4(b), where $i = 4$). All paths starting from either of these vertices can visit only vertices whose indices are strictly larger than $i+1$. Therefore, there is no path from i to $i+1$ nor from $i+1$ to i, implying that G is not semi-connected, a contradiction.

We now use this observation to provide us with an algorithm. First, apply topological sort to G, and label the vertices of the graph as $V = \{1, \ldots, n\}$ according to this ordering. Next, check that every pair $(i, i + 1) \in E$ for $1 \leq i < n$. If so, output "yes" and otherwise output "no". This can be done in $O(n + m)$ time, by the topological ordering algorithm given in class, followed by a DFS to verify the existence of these edges.

Solution 5:

(a) An example is shown in Fig. 5, where s is the source vertex.

- The algorithm first processes s, which sets $d[a] = 2$ and $d[b] = 4$.
- Next is a at distance $d[a] = 2$. This performs $\mathsf{relax}(a, c)$, which sets $d[c] = 3$.
- Next is c at distance $d[c] = 3$, which does nothing, since c has no outgoing edges.

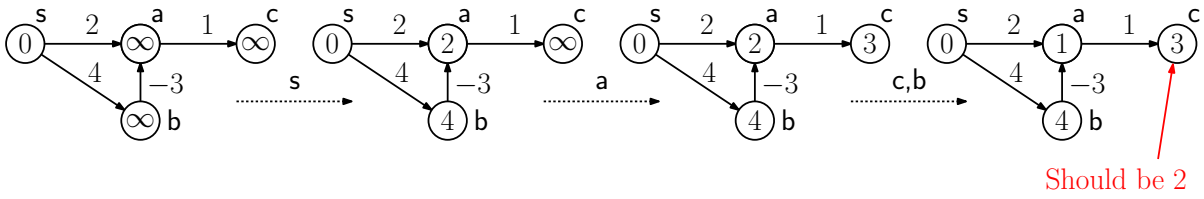

Figure 5: Why Dijkstra fails on graphs with negative edge weights.

- Finally, is b at distance $d[b] = 4$. This performs relax(b, a), which sets $d[a] = 1$.

Observe that on termination, $d[c] = 3$, but in fact $\delta(s, b) = 2$ by the path $\langle s, b, a, c\rangle$. (Note that this assumes the version of Dijkstra's algorithm presented in class, which performs the relax operator on all outgoing edges, not just the edges to unprocessed vertices. Different variants of Dijkstra's algorithm will have different counterexamples.)

(b) We will do this by recalling the proof of Dijkstra's algorithm given in the lecture, and showing that these negative edges do not affect the correctness proof.

The crux of the correctness proof for Dijkstra is showing that, whenever a vertex u is processed, $d[u]$ is equal to the true distance $\delta(s, u)$. Suppose to the contrary that this is not true, and consider the *first* finished vertex u for which this fails to hold, which implies that $d[u] > \delta(s, u)$.

Let S denote the set of processed vertices, just prior to processing u. The true shortest path from s to u must exit S along some edge (x, y), where $x \in S$ and $y \notin S$ (see Fig. 6).

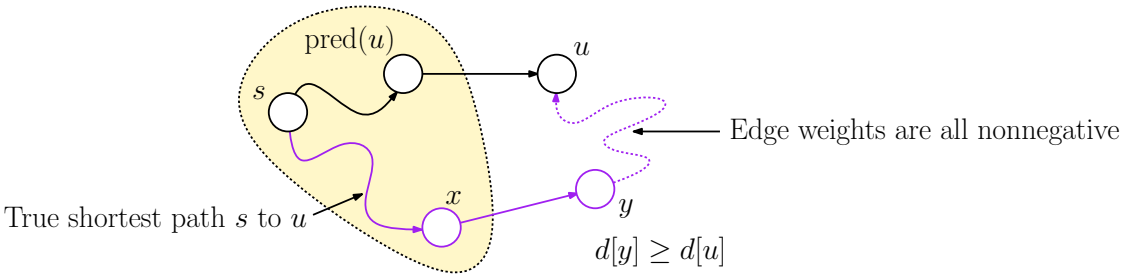

Figure 6: Correctness of Dijkstra's Algorithm even with negative edges from the source.

Because u is the first vertex where we made a mistake, and since x was already processed, we have $d[x] = \delta(s, x)$. When the algorithm processes x, it performs relax(x, y), which implies that

$$d[y] \;=\; d[x] + w(x, y) \;=\; \delta(s, x) + w(x, y) \;=\; \delta(s, y).$$

Observe that the remainder of the path from y to u will not revisit s, since if that were to happen, it would mean that there is a negative cost cycle from s through y and back to s. Since we do not revisit s, all the edges along the path from y to x are nonnegative. Therefore, $\delta(s, y) \le \delta(s, u)$. Because u was chosen before y for processing, we have $d[u] \le d[y]$. Putting this together, we have

$$\delta(s, u) \;<\; d[u] \;\le\; d[y] \;=\; \delta(s, y) \;\le\; \delta(s, u).$$

Clearly we cannot have $\delta(s,u) < \delta(s,u)$, which establishes the desired contradiction. Therefore, Dijkstra's algorithm is correct.

An alternative argument is based on the idea that we can transform G into a graph that has no negative weight edges, but behaves exactly in the same manner from the perspective of Dijkstra's algorithm. First, let's assume that there is at least one negative weight edge emanating from s, for otherwise, all the edge weights are nonnegative, and Dijkstra's algorithm is correct. Let $-W < 0$ denote the smallest weight among all the edge emanating from s. Consider a modified graph G' which results by increasing the weights of all edges emanating from s by $+W$ (see Fig. 7). Let G' denote the resulting graph. Now, when we initialize Dijkstra's algorithm, we compensate for the change by setting $d[s] = -W$. Observe that the costs of all the paths in G' are exactly the same as in G. Furthermore, since all the edges now have nonnegative weights, Dijkstra's algorithm operates on a graph with nonnegative edge weights, and hence is correct.

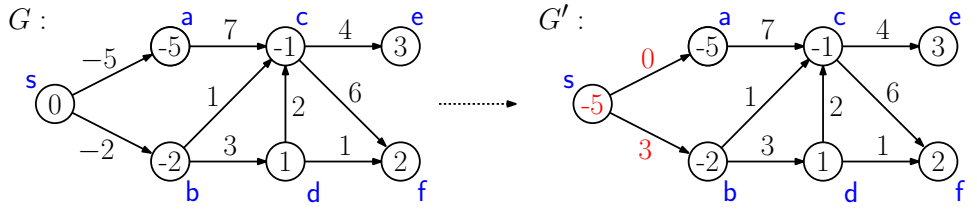

Figure 7: Transforming a graph into an equivalent one with nonnegative edge weights.