ECE608 CHAPTER 22 PROBLEMS

1) 22.1-6
2) 22.2-6
3) 22.2-7
4) 22.3-7
5) 22.3-8
6) 22.4-3
7) 22.4-5
8) 22.5-1
9) 22.5-3
10) 22.5-7
11) 22-1

(1) CLR 22.1-6

We are looking for a universal sink i.e., a vertex with in degree of $|V| - 1$ and out degree of zero. In the adjacency matrix the elements in row i represent the edges that our out-going from node i to other nodes. And the elements in column j represent the edges that are in-coming to node j. We are looking for k, such that the k'th row has all zero's and the k'th column has all 1's except for the element in the k'th row (which is a zero).

Start traversing the first row and stop at the first 1 that is encountered. This will take O(V) steps in the worst case. If no 1 is encountered then this row is the only possible candidate for a universal-sink (because it does not have an edge to any other node, no other node can be a universal sink), we can simply check by traversing the first column in O(V) time and see if it has all 1's. If so then node 1 is a universal sink otherwise the graph has no universal sink. Note that the algorithm terminates once we find a row of all zero's whether that row represents a universal-sink or not, thus guaranteeing O(V) running time.

Now say that while traversing the first row we encounter the first 1 at column k. Then the elements 0 through k-1 in the first row have zeros. This means that no node in 0 through k-1 can be universal-sink's because node 1 does not have an edge to any of them. SO we have effectively eliminated k-1 nodes from the possible candidates. In addition we have also eliminated node 1 because its row is not all zero's. We did this elimination in O(k) steps by examining the first row.

Now, out of the un-eliminated nodes, we begin examining the next available row j (this may not be the second row of the adjacency matrix) and follow a procedure similar to the first row. If no 1 is encountered we test the j'th column for 1's and terminate as explained above. If a 1 is encountered after m zero's then againwe have eliminated m-1 nodes plus node j itself from being a universal sink. In total we have eliminated k+m nodes in O(k+m) time.

Repeating the above procedure, examimining one row at a time those vertices that have not yet been eliminated, we can find whether a universal-sink exists or not. We will require in the worst case O(V) time before all nodes will have been eliminated. Thus algorithm is guaranteed to terminate in O(V) time.

(2) CLR 22.2-6

See Figure 1. The two child nodes of the root s are both discovered in the first iteration of BFS. Whichever of the two nodes BFS chooses to traverse first, in the next iteration both the remaining nodes will become children of that node. Therefore it is impossible for BFS to obtain the shortest path tree shown in Figure 1.

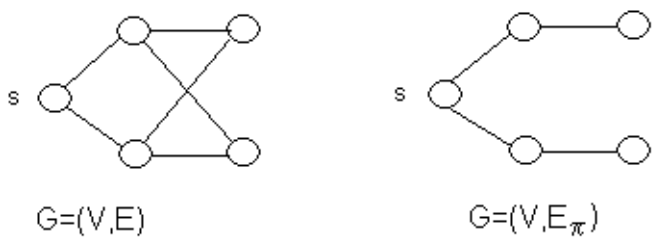

Figure 1: A graph $G = (V, E)$ and a corresponding shortest-path tree $G' = (V, E_\pi)$, such that G' cannot be obtained by running BFS on G

(3) CLR 22.2-7

The first step to solve this problem is treat each wrestler as vertex in an undirected graph, and rivalry between a pair of wrestlers as an edge connecting vertices representing the pair of wrestlers. We call such graph G. We can then determine if G is a bipartite graph (see section B.4, p. 1083 for the definition of a bipartite graph) by modifying BFS algorithm.

For a graph to be bipartite, it must be undirected and there must be a way of partitioning the vertices into two groups such that there is no edge between nodes that are in the same group (i.e., all edges are between vertices in the two groups). For this to be violated, there would have to be an edge between two nodes that are at the same distance from a source vertex. In order to check all components not connected to a particular, we would need to add an outer loop to check all components for this property. If G is a bipartite graph, then we can label the wrestlers as good guys and bad guys such that each rivalry is between a good guy and a bad guy.

An alternative form of this is to run BFS over all vertices and then determine whether there is an edge between any two vertices such that their distances from that current source vertex are either both even or both odd. In this case, the graph is not bipartite. Let the evens be good guys and the odds be bad guys.

Below is an algorithm that labels vertices as GOOD or BAD, and then determines whether there are any edges between two good guys or two bad guys. If so, then there is no designation and we fail; otherwise, we will have a designation of wrestlers as one or the other.

GoodBadGuys(list of wrestlers, pairs of rivalries)

1. Build an adjacency list representation for $G = (V, E)$ such that each wrestler corresponds to a vertex in V and each edge in $E = (V_1, V_2)$ represents a rivalry between the wrestlers associated with V_1 and V_2
2. **for** each vertex $u \in V[G]$
3. **do** $color[u] \leftarrow$ WHITE
4. **for** each vertex $u \in V[G]$
5. **do if** $color[u] =$ WHITE
6. **then** $guy \leftarrow$BFS-GOOD-BAD(G, u, guy)
7. return guy

BFS-GOOD-BAD(G, s, guy)

1. $color[s] \leftarrow$ Gray
2. $guy[s] \leftarrow$ GOOD
3. $Q \leftarrow \emptyset$
4. ENQUEUE(Q, s)
5. **while** $Q \neq \emptyset$
6. **do** $u \leftarrow$ DEQUEUE(Q)
7. **for** each $v \in Adj[u]$
8. **do if** $color[v] =$ GRAY and $guy[v] = guy[u]$
9. **then return** FAIL
10. **if** $color[v] =$ WHITE
11. **then** $color[v] \leftarrow$ GRAY
12. **if** $guy[u] =$ GOOD
13. **then** $guy[v] \leftarrow$ BAD
14. **else** $guy[v] \leftarrow$ GOOD
15. ENQUEUE(Q, v)
16. $color[u] \leftarrow$ BLACK
17. **return** guy

(4) CLR 22.3-7

DFS-STACK(G)

1. **for** each vertex $u \in V[G]$
2. **do** $color[u] \leftarrow$ WHITE
3. $\pi[u] \leftarrow$ NIL
4. $time \leftarrow 0$
5. **for** each vertex $u \in V[G]$
6. **do if** $color[u] =$ WHITE
7. **then** DFS-VISIT-STACK(u)

DFS-VISIT-STACK(u)

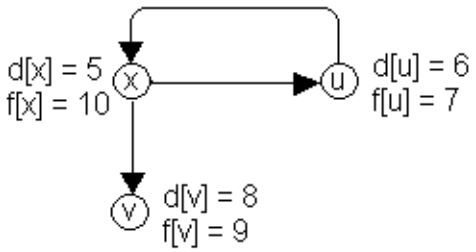

Figure 2: A counter example showing a graph in which there is a path from u to v but v does not become a descendant of u

```
1. PUSH(S, u)
2. while !empty(S)
3.         u ← POP(S)
4.         if color[u] = WHITE
5.           then color[u] ← GRAY
6.                PUSH(S, u)
7.                time ← time + 1
8.                d[u] ← time
9.                for each v ∈ Adj[u]
10.                    do if color[v] = WHITE
11.                       then π[v] ← u
12.                            PUSH(S, v)
13.          else color[u] ← BLACK
14.               f[u] ← time ← time + 1
```

(5) CLR 22.3-8

There may be a path from u to v but some node along that path may be non-WHITE. This means that when DFS-VIST is called on u we never arrive at v because there is no WHITE path from u to v. In Figure 2 u has a path to v through an ancestor node x but when x is discovered during DFS-VISIT(u) x is colored GRAY so v is never discovered during DFS-VISIT(u).

(6) CLR 22.4-3

Graph has a cycle if and only if we encounter a back edge pointing to a vertex that we have already visited (i.e., a gray node).

```
DFS(G)
1. for each vertex u ∈ V[G]
2.     do color[u] ← WHITE
3.        π[u] ← NIL
4. time ← 0   ▷ global timestamp
5. for each vertex u ∈ V[G]
6.     do if color[u] = WHITE
7.           then DFS-Visit-Findcycle(u)
```

```
DFS-Visit-Findcycle(u)
1.  color[u] ← Gray
2.  d[u] ← time ← time+1
3.  for each v ∈ Adj[u]
4.      do if (color[v] = Gray and v ≠ π(u))
5.            then return "cycle found"
6.         if color[v] = White
7.            then π[v] ← u
8.                 DFS-Visit-Findcycle(v)
9.  color[u] ← Black
10. f[u] ← time ← time+1
```

Note: For an undirected graph:

(i) Connected tree without a cycle with V nodes contains exactly $V - 1$ edges

(ii) If we add any edges to a tree without a cycle, it must form a cycle.

Since the **for** loop in DFS-Visit-Findcycle is terminated as soon as one cycle is found or we process $V - 1$ edges, the running time is $O(V)$.

(7) CLR 22.4-5

```
TOPOLOGICAL-SORT(G)
1. for each vertex u ∈ V          ▷ Initialize in-degree, Θ(V) time
2.     do in-degree[u] ← 0
3. for each vertex u ∈ V          ▷ Compute in-degree, Θ(V + E) time
4.     do for each v ∈ Adj[u]
5.           do in-degree[v] ← in-degree[v] + 1
6. ▷ Initialize Queue, Θ(V) time
7. Q ← 0
8. for each vertex u ∈ V
9.     do if in-degree[u] = 0
10.       then ENQUEUE(Q, u)
11. ▷ while loop takes O(V + E) time
```

```
12. while Q ≠ 0
13.         do u ← DEQUEUE(Q)
14.             output u
15.             ▷ for loop executes O(E) times in total
16.             for each v ∈ Adj[u]
17.                 do in-degree[v] ← in-degree[v] − 1
18.                     if in-degree= 0
19.                     then ENQUEUE(Q, v)
20. ▷ Check for cycles, O(V) time
21. for each vertex u ∈ V
22.     do if in-degree[u] ≠ 0
23.         then report that there's a cycle
24. ▷ Another way to check for cycles would be to count the vertices
25. ▷ that are output and report a cycle if that number is < |V|.
```

To find and output vertices of in-degree 0, we first compute all vertices' in-degrees by making a pass through all the edges (by scanning the adjacency lists of all the vertices) and incrementing the in-degree of each vertex an edge enters. This takes $\Theta(V+E)$ time ($|V|$ adjacency lists accessed, $|E|$ edges total found in those lists, $\Theta(1)$ work for each edge).

We keep the vertices with in-degree 0 in a FIFO queue, so that they can be enqueued and dequeued in $O(1)$ time. (The order in which vertices in the queue are processed doesn't matter, so any kind of queue works.) Initializing the queue takes one pass over the vertices doing $\Theta(1)$ work, for total time $\Theta(V)$.

As we process each vertex from the queue, we effectively remove its outgoing edges from the graph by decrementing the in-degree of each vertex one of those edges enters, and we enqueue any vertex whose in-degree goes to 0. There's no need to actually remove the edges from the adjacency list because that adjacency list will never be processed again by the algorithm: Each vertex is enqueued/dequeued at most once because it is enqueued only if it starts out with in-degree 0 or if its in-degree becomes 0 after being decremented (and never incremented) some number of times.

(i) A vertex is removed from the queue $O(V)$ times because no vertex can be enqueued more than once. The per-vertex work (dequeue and output) takes $O(1)$ time, for a total of $O(V)$ time.

(ii) Because the adjacency list of each vertex is scanned only when the vertex is dequeued, the adjacency list of each vertex is scanned at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, at most $O(E)$ time is spent in total scanning adjacency lists. For each edge in an adjacency list, $\Theta(1)$ work is done, for a total of $O(E)$ time.

Thus, the total time taken by the algorithm is $O(V + E)$. The algorithm outputs vertices in the right order (u before v for every edge (u, v)) because v will not be output until its in-degree becomes 0, which happens only when every edge (u, v)

leading into v has been "removed" due to the processing (including input) of u. If there are no cycles, all vertices are output.

Proof: Assume that some vertex v_0 is not output. v_0 cannot start out with in-degree 0 (or it would be output), so there are edges into v_0, and v_0's in-degree never becomes 0, so at least one edge (v_1, v_0) is never removed, which means that at least one other vertex v_1 was not output. Similarly, v_1 not output $\rightarrow$ some vertex v_2 such that $(v_2, v_1) \in E$ was not output, ... Since the number of vertices is finite, this path $\ldots \rightarrow v_2 \rightarrow v_1 \rightarrow v_0$ is finite, so we must have $v_i = v_j$ for some i and j in this sequence, which means there is a cycle.

If there are cycles, not all vertices will be output, because some in-degrees never become 0.

Proof: Assume that a vertex in a cycle is output (its in-degree becomes 0). Let v be the first vertex in its cycle to be output, and let u be v's predecessor on the cycle. In order for v's in-degree to become 0, the edge (u, v) must have been "removed," which happens only when u is processed. But this can't have happened, because v is the first vertex in its cycle to be processed. Thus no vertices in cycles are output.

(8) CLR 22.5-1

If the newly added edge belongs to one of the original strongly connected components, then no new strongly connected component will be added. If the newly added edge connects two vertices that belong to two different strongly connected components, it is possible that those two or more strongly connected components can merge into one strongly connected component. Thus, the number of strongly connected components will be the same or less if a new edge is added.

(9) CLR 22.5-3

The professor is incorrect. Depending on the order in which children are considered during Depth-first Search, it is possible for a vertex to finish first in a strongly connected component that has edges leading out to other strongly connected components. Consider the graph $G = (V, E)$ with $V = \{A, B, C\}$ and $E = \{(A, B), (B, A), (A, C)\})$. This graph has two SCCs, $\{A, B\}$ and $\{C\}$. Depth-first search started at A may result in two possible finish-time orderings, depending on whether B or C is considered first among the children of A. If B is considered first, then B will finish first. In that case, a second depth-first search on G will color all three vertices in a single call to DFS-VISIT starting from the earliest finisher B, even though all three vertices are not in the same SCC.

(10) CLR 22.5-7

First run STRONGLY-CONNECTED-COMPONENTS on graph G to obtain the SCC components of the graph. Collapse each strongly connected component into a single vertex and obtain the graph G^{SCC} as shown on pg 553 of CLR. Now call

TOPOLOGICAL-SORT on the graph G^{SCC}, we know that because G^{SCC} is a directed acyclic graph we will correctly obtain a unique linear ordering of all the strongly connected components. Now if a component C_i appears earlier in the list than another component C_j, then it must be the case that C_i is not reachable from C_j (due to the topological ordering of an acyclic graph). Then, for the property of semi-connectedness to hold we require that C_j must be reachable by C_i, and this property must hold true for every such C_i and C_j (i.e., such that $i < j$). Because we must guarantee this for every pair, it is necessary and sufficient to show that this reachability holds for every (C_i, C_{i+1}) pair of components. This can be easily checked by traversing the entire topological order of G^{SCC} in $O(V)$ time in the worst case because the topological linear ordering is $O(V)$ long. The running time for STRONGLY-CONNECTED-COMPONENTS and TOPOLOGICAL-SORT are both $O(V + E)$. So the overall running time for our algorithm is $O(V + E)$

(11) CLR 22-1

(a)

1. If u and v are unrelated by ancestry, then (u, v) is a cross edge. For all remaining edges, u and v are related by ancestry.
   Because it is a breadth-first search of an undirected graph, for any edge (u, v), if v is a descendant of u, then this edge must be explored immediately when we are discovering the immediate successors of u, and so it can only be a tree edge, not a forward edge.
   If for any edge (u, v), v is an ancestor of u, then this edge must be explored when we are discovering the immediate successors of v, and so it can only be a tree edge, not a back edge.
   Therefore, there are no back edges or forward edges.
2. For each tree edge (u, v), because this edge is explored while we are discovering the immediate successors of u, by the definition of $d[], d[v] = d[u] + 1$. Once this value is set, it cannot be changed.
3. For any edge (u, v), we know that $\delta(s, v) \leq \delta(s, u) + 1$. By Theorem 23.4 proved on p. 473 of the CLR textbook, we also know that $d[v] = \delta(s, v)$ and $d[u] = \delta(s, u)$ at the end of BFS. Hence, it follows that $d[v] \leq d[u] + 1$, thus it is necessary that $d[v] - d[u] \leq 1$ if (u, v) is an edge in G.
   Note that because the edge (u, v) is a cross edge, it must be the case that v is on the queue when (u, v) is explored; otherwise, (u, v) would be a tree edge. By Lemma 23.3 on p. 473 of CLR we know that Q contains vertices in the order $(v_1, v_2, \ldots, v_r)$ and that $d[v_r] \leq d[v_1] + 1$, and for all i going from 1 to $r - 1$, $d[v_i] \leq d[v_{i+1}]$. Thus, $d[v_1] \leq d[v_2] \leq \ldots \leq d[v_r] \leq d[v_1] + 1$. When the cross edge is discovered, u is at the head of the queue, and so either $d[v] = d[u]$ or $d[v] = d[u] + 1$.
   Because BFS operates a level at a time, the cross edges will always be discovered at the vertex in (u, v) which is closer to the front of the queue, namely u. If u and v have the same d values, they are descendants of a common vertex, so when the cross edge is discovered, $d[u] = d[v]$. If u and

v have different d values, then the vertex with the smaller value will be explored first (namely u), at which point BFS will discover the cross edge and $d[v] = d[u] + 1$.

(b)

1. In a breadth-first search of a directed graph, for any edge (u, v), if v is a descendant of u, then this edge must be explored immediately when we are discovering the immediate successors of u, and so it can only be a tree edge, not a forward edge.
2. For each tree edge (u, v), because this edge is explored while we are discovering the immediate successors of u, by the definition of $d[], d[v] = d[u] + 1$. Once this value is set, it cannot be changed.
3. For any edge (u, v), we know that $\delta(s, v) \leq \delta(s, u) + 1$. By Theorem 23.4 proved on p. 473 of the CLR textbook, we also know that $d[v] = \delta(s, v)$ and $d[u] = \delta(s, u)$ at the end of BFS. Hence, it follows that $d[v] \leq d[u] + 1$.
4. If (u, v) is a back edge, there must be one or more tree edges going from v to u, since v is an ancestor of u. As each edge must increase the cost of $d[u]$ by adding 1 to the cost of $d[v]$, it follows that $d[v] < d[u]$. Since all edge weights are 1 and $d[v]$ could be the source vertex, it follows that $0 \leq d[v]$. Hence, $0 \leq d[v] < d[u]$.