Spring 2016
ECE608, Homework #10 Solution

(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.

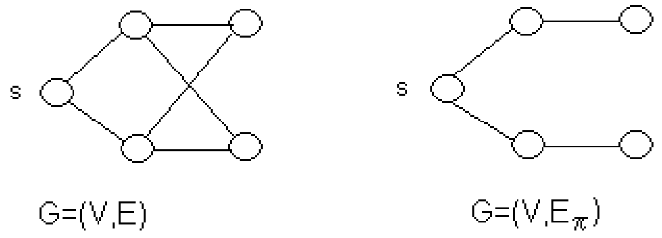


Figure 1: A graph $G = (V, E)$ and a corresponding shortest-path tree $G' = (V, E_\pi)$, such that G' cannot be obtained by running BFS on G

(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 22-1

(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, $0 \leq d[v] < d[u]$.

(5) 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, $0 \leq d[v] < d[u]$.

(6) CLR 24.1-3

The proof of Lemma 24.2 shows that for every v, $d[v]$ has attained its final value after *length* (any shortest-weight path to v) iterations of BELLMAN-FORD. Thus after m passes, BELLMAN-FORD can terminate. We don't know m in advance, so we can't make the algorithm loop exactly m times and then terminate. But if we just make the algorithm stop when nothing changes any more, it will stop after $m+1$ iterations (i.e., after one iteration without a change), unless there is a negative weight cycle. Hence, we should also make sure that the number of iterations does not exceed $|V[G]| - 1$.

```
BELLMAN-FORD-$(M+1)(G, w, s)$
1. INITIALIZE-SINGLE-SOURCE$(G, s)$
2. $changes \leftarrow$ TRUE
3. $iteration \leftarrow 1$
4. while $(changes =$ TRUE$) \wedge (iteration \leq |V[G]| - 1)$
5.        do $changes \leftarrow$ FALSE
6.           $iteration \leftarrow iteration + 1$
7.           for each edge $(u, v) \in E[G]$
8.                do RELAX-M$(u, v, w)$
9. for each edge $(u, v) \in E[G]$
10.     do if $d[v] > d[u] + w(u, v)$
11.           then return FALSE
12. return TRUE
```

```
RELAX-M$(u, v, w)$
1. if $d[v] > d[u] + w(u, v)$
2.    then $d[v] \leftarrow d[u] + w(u, v)$
3.         $\pi[v] \leftarrow u$
4.         $changes \leftarrow$ TRUE
```

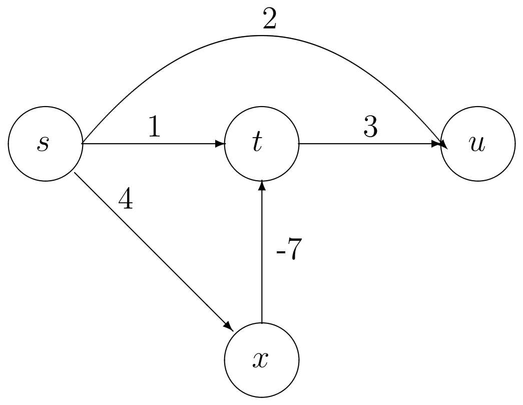
(7) CLR 24.1-4

Change line 7 of the BELLMAN-FORD algorithm, as given in the CLR text, to:

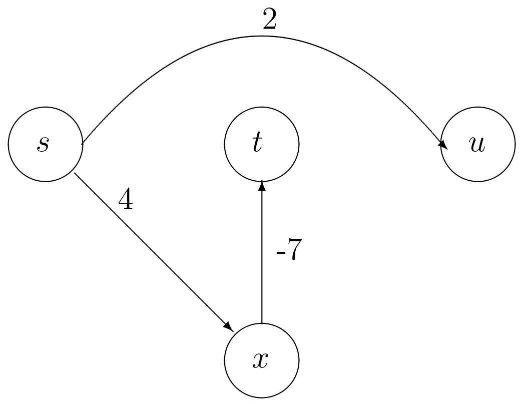
then $d[v] \leftarrow -\infty$

(8) CLR 24.3-2

Consider the graph below:



The predecessor subgraph returned by DIJKSTRA's algorithm will look like this:



The correct answer should use the path $s \rightarrow x \rightarrow t \rightarrow u$ instead of $s \rightarrow u$, since $\delta(s,u) = 0$.

In Theorem 24.6, the assumption of no negative weights is used to conclude that $\delta(s,y) \leq \delta(s,u)$ and $d[y] \leq d[u]$. If there are negative weights in the path p, $\delta(s,y)$ may be greater than $\delta(s,u)$, and the contradiction will not hold, as $d[u]$ can be greater than $\delta(s,u)$ and hence $d[u] \neq \delta(s,u)$.

(9) CLR 24.3-8

We use the loop invariant:

At the start of each iteration of the while loop of lines 4-8, $d[v] = \delta(s,v)$ for each vertex $v \in S$.

Initialization: Initially $S = \phi$ so the invariant is trivially true.

Maintenence: To prove by contradiction, assume that u is the first vertex added to S such that $d[u] \neq \delta(s,u)$. We know that $u \neq s$ because s is the first vertex added to S and $d[s] = \delta(s, s) = 0$ at that time. As $u \neq s$ we know that $S \neq \phi$ when u is added.

There is a shortest-path p from s to u in G, otherwise $d[u] = \delta(s, u) = \infty$ from the very beginning, contradicting our initial assumption. Say on path p, when going from s to u, y is the first vertex in $V - S$. Also, x is the predecessor of y along this path.

$d[y] = \delta(s, y)$ when u is added to S. To see why this is so we observe that x was added to S before u. ANd when x was added to S the edge (x, y) was relaxed and since $d[x] = \delta(s, x)$ at that moment, therefore $d[y]$ gets set to $\delta(s, y)$.

No edge along the path from y to u has anegative weight because all negative edges must be of the form $(s, -)$ and $s \neq y$. Because y occurs before u on a path with no negative-weight edges $\delta(s, y) \leq \delta(s, u)$. Then we have

$$d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$$

Now both y and u were in $V - S$ when u was chosen as the next vertex to add to S therefore

$$d[u] \leq d[y]$$

From the above two equations we have $d[u] = \delta(s, u)$ contradicting our intitial assumption.

Termination: $Q = \phi$ and $S = V$, thus $d[v] = \delta(s, v)$ for all $v \in S$ and the shortest path has been solved.