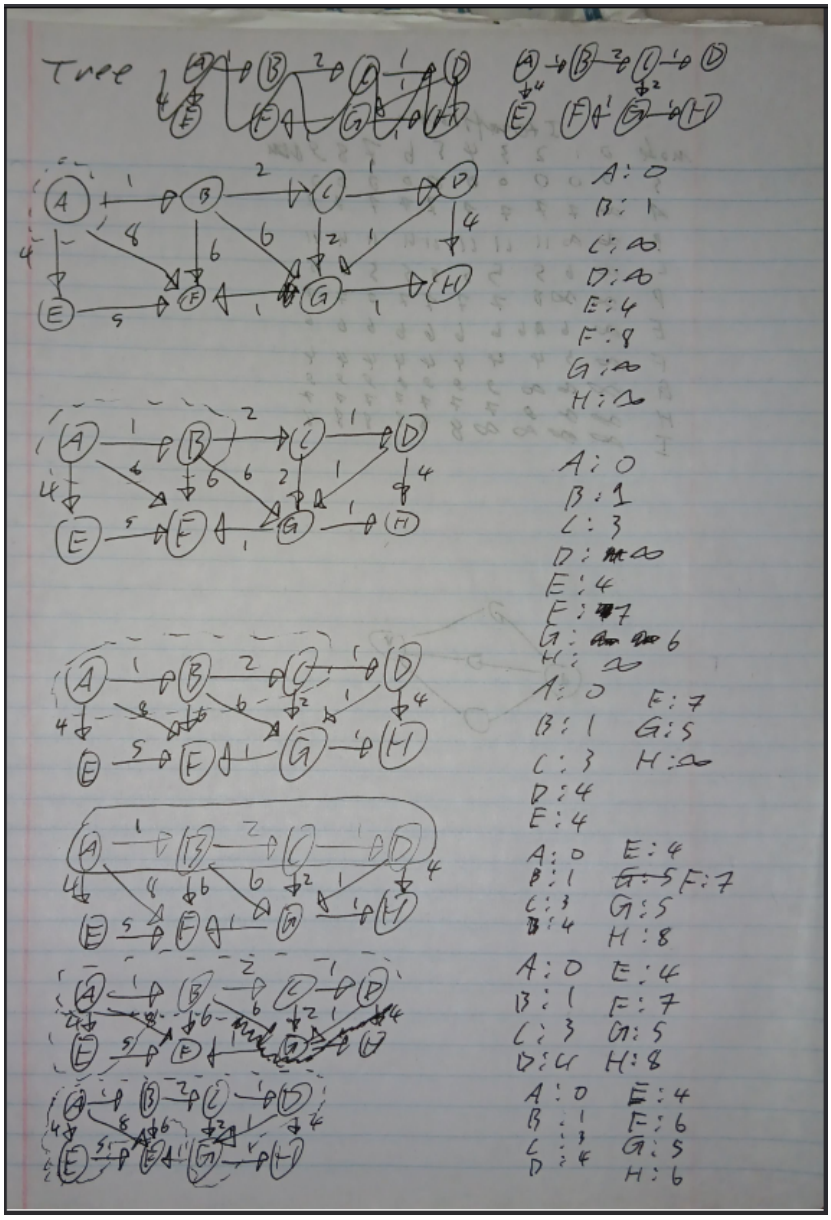
3.25 (a) Input: $G = (V, E), v \in V$
Output: cost

for $u \in V$ **do** $minCost = p_u$

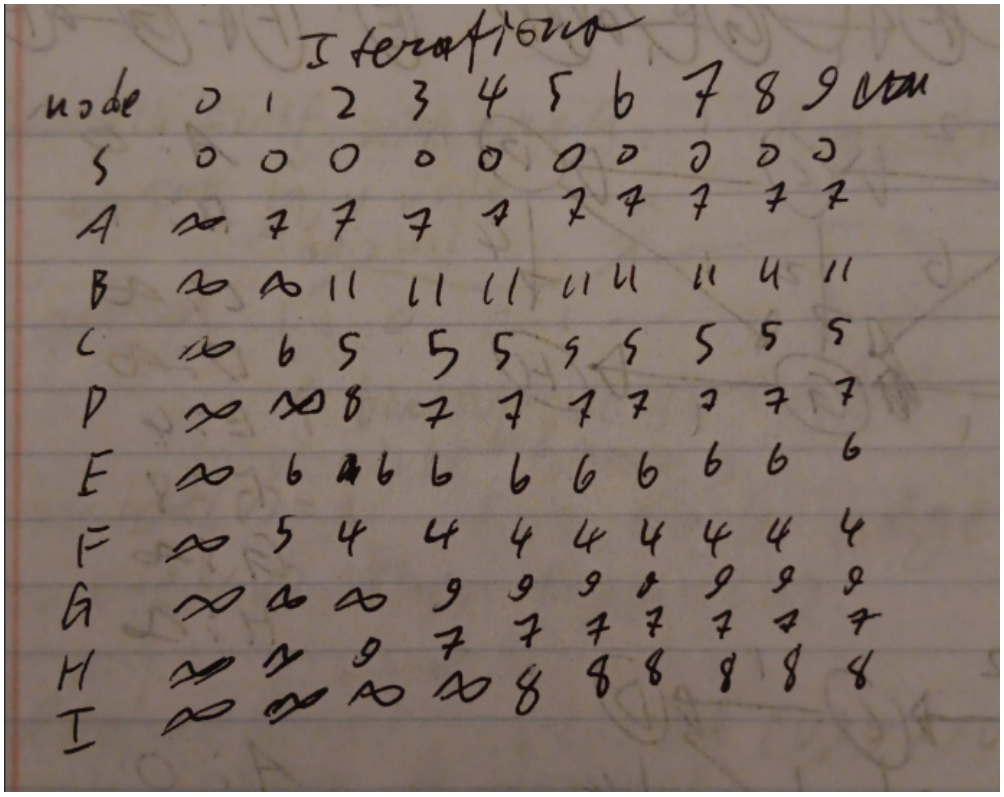
for $(v, u) \in$ **do**
minCost $= \min minCost, p_v$
end for
end for if not visited

(b) For a general digraph, one can first split up the digraph into it's strongly connected components, just find the maximum cost per strongly connected node, then run the algorithm as mentioned above on this newly formed graph. This should take linear time.

4.1


Tree
A: 0
B: 1
C: ∞
D: ∞
E: 4
F: 8
G: ∞
H: ∞
A: 0
B: 1
C: 3
D: ∞
E: 4
F: 7
G: 6
H: ∞
A: 0 E: 7
B: 1 G: 5
C: 3 H: ∞
D: 4
E: 4
A: 0 E: 4
B: 1 F: 7
C: 3 G: 5
D: 4 H: 8
A: 0 E: 4
B: 1 F: 7
C: 3 G: 5
D: 4 H: 8
A: 0 E: 4
B: 1 F: 6
C: 3 G: 5
D: 4 H: 6

4.2



4.5 Input Graph $G = (V, E)$, nodes $u, v \in V$

Output minCons, the number of times v is reached by a shortest connection

for all $w \in V$ **do**:
    seen(w) = $false$
end for
dist(u) = true
seenv = false
minCons = 0
$Q = [u]$

while Q is not empty **do**:
    w = eject (Q)
    for all edges $(w, x) \in E$ **do**:
        if seen(x) = false **then**
            if $x = v$ **then**
                minCons++
                seenv = true
            else
                seen(x) = true
            end if
            if not seenv **then**
                inject(Q, w)
            end if
        end if

end for
end while

Above is a modified version of BFS where we don't care about distance. Since BFS will have equal distance to every node when spreading out (unless it's a leaf and v isn't encountered), all we care about is the first (minimal) encounter with v, then from that point we stop putting nodes on the stack and check if any other nodes in the same distance will reach v, then we return the number of times this occurs, giving us the number of equivalent minimal paths from u to v.

chatGPT Input Graph $G = (V, E)$, source $s \in V$
Output whether or not the graph is bipartite or not

for all $u \in V$ **do**
color(u) = Null

end for
color(s) = true
$Q = [s]$

while Q is not empty **do**
u = Eject(Q)
for $(u, v) \in E$ **do**
if color(v) = Null **then**
color(v) = not color(u)
inject(Q, v)
else if color(v) = color(u) **then**
return false
end if
end for
end while