3.25

a) Algorithm for directed acyclic graphs:

1. Topologically sort the vertices of the DAG using DFS
2. Initialize an array cost with size $|V|$ and set all elements to infinity, except for the source node which is set to its own price
3. Traverse the vertices in topological order
4. For each vertex u in topological order:
   - Update the cost [u] to the minimum of its current value and the cost of its neighbors plus the price of u
5. The cost array will now contain the cost values of all vertices

Time Complexity: $O(V+E)$, V is the number of vertices, E is the number of edges

b) Algorithm for all directed graphs:

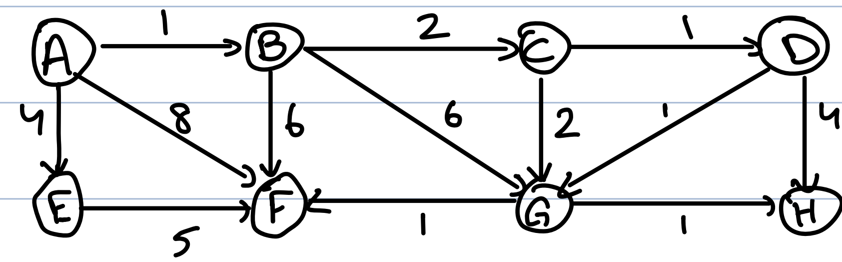
1. Find the strongly connected components of the graph G.
2. Create a new DAG G' where each strongly connected component in G is represented as a single node
3. Compute the cost array for the new DAG G' using part a's algorithm
4. Iterate through the strongly connected components in the topological order of G and update the cost values of the vertices within each strongly connected component using the computed cost values for the strongly connected component itself to distribute the cost values back to

the original vertices

5. The cost values for each vertex in the original graph will now be present in the cost array.

Time Complexity: $O(V+E)$, V and E are the number of vertice and edge counts, respectively.

4.1



a)

Iteration 1: Visit A

Unvisited nodes: {B,C,D,E,F,G,H}

Distances:

A: 0
B: 1
C: ∞
D: ∞
E: 4
F: 8
G: ∞
H: ∞

Iteration 2: Visit B

Unvisited nodes: {C,D,E,F,G,H}

Distances:

A: 0
B: 1
C: 3
D: ∞
E: 4
F: 7
G: 7
H: ∞

Iteration 3: Visit C

Unvisited nodes: {D,E,F,G,H}

Distances:

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

Iteration 4: Visit D

Unvisited nodes: $\{E, F, G, H\}$

Distances:

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

Iteration 5: Visit E

Unvisited nodes: $\{F, G, H\}$

Distances:

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

Iteration 6: Visit G

Unvisited nodes: $\{F, H\}$

Distances:

A: 0
B: 1
C: 3
D: 4
E: 4
F: 6
G: 5
H: 6

Iteration 7: Visit F

Unvisited nodes: $\{H\}$

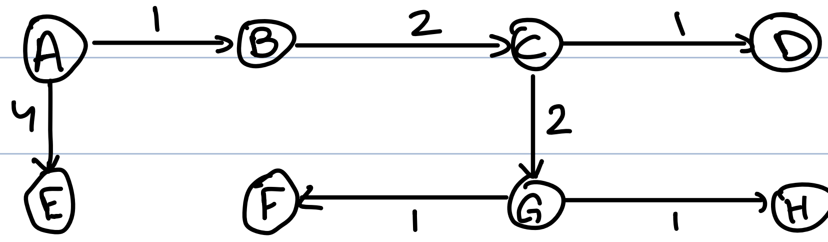
Distances:

A: 0
B: 1
C: 3
D: 4
E: 4
F: 6
G: 5
H: 6

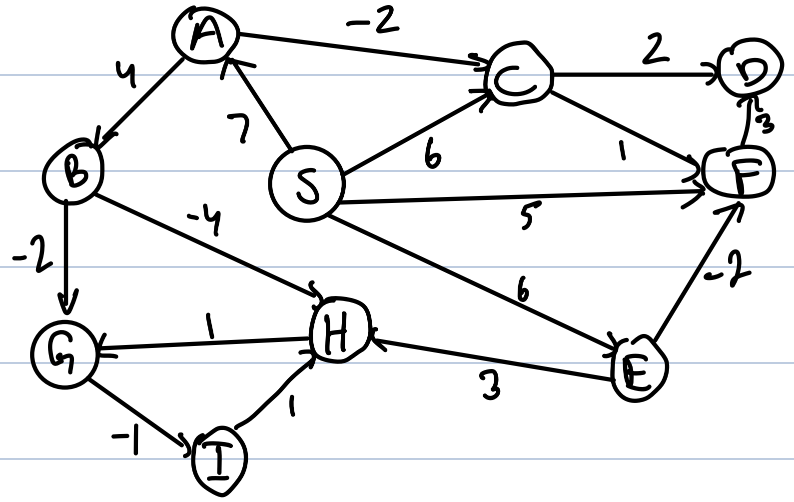
No available vertices to visit from F or H.

Iteration	A	B	C	D	E	F	G	H
0	0	∞	∞	∞	∞	∞	∞	∞
1	0	1	∞	∞	4	8	∞	∞
2	0	1	3	∞	4	7	7	∞
3	0	1	3	4	4	7	5	∞
4	0	1	3	4	4	7	5	8
5	0	1	3	4	4	7	5	8
6	0	1	3	4	4	6	5	6

b) Final shortest path tree:



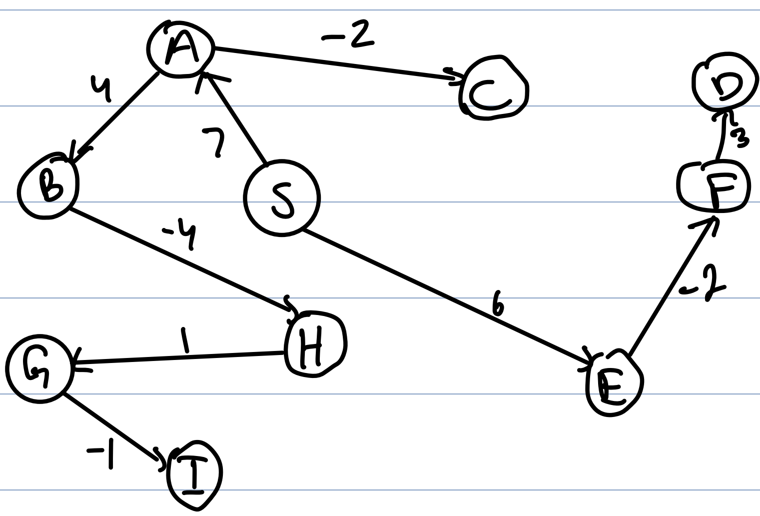
4.2



a)

Iteration	S	A	B	C	D	E	F	G	H	I
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	7	∞	6	∞	6	5	∞	∞	∞
2	0	7	11	5	8	6	4	∞	9	∞
3	0	7	11	5	7	6	4	9	7	∞
4	0	7	11	5	7	6	4	8	7	8
5	0	7	11	5	7	6	4	8	7	7
6	0	7	11	5	7	6	4	8	7	7

b)



4.5 In order to find the number of distinct shortest paths from u to v, we can use a BFS algorithm that can also count the number of shortest paths.

```
def count_paths(graph, u, v):
    queue = dequeue()
    dist = {}
    count = {}
    queue.append(u)
    dist[u] = 0
    count[u] = 1
    while queue:
        x = queue.popleft()
        for neighbor in graph[x]:
            if neighbor not in dist:
                dist[neighbor] = dist[x] + 1
                queue.append(neighbor)
                count[neighbor] = count[x]
            elif dist[neighbor] == dist[x] + 1:
                count[neighbor] += count[x]
    return count[v] if v in count else 0
```

This algorithm's time complexity is O(V+E), V is the number of vertices, E is the number of edges

344 helper:

```
def is_bipartite(graph):
    visited = {}
    queue = deque()
    start = next(iter(graph.keys()))
    queue.append(start)
    visited[start] = 'A'
    while queue:
        current = queue.popleft()
        for neighbor in graph[current]:
            if neighbor not in visited:
                visited[neighbor] = 'B' if visited[current] == 'A' else 'A'
                queue.append(neighbor)
            elif visited[neighbor] == visited[current]:
                return False
    return True
```