Homework 5

Name:	Rohan Karamel
NetID:	rak218
Course:	Algorithms Section 2
Instructor:	Professor Mario Szegedy
Date:	April 3, 2024

Problem 3.25a. Give a linear-time algorithm that works for directed acyclic graphs.

Solution The algorithm is as follows:

- The algorithm begins by performing a topological sort on the graph G. This is a linear-time operation that provides an ordering of the vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. The result of the topological sort is stored in the array T.
- The algorithm then iterates over the vertices in reverse topological order. For each vertex a, it initializes a variable min to the cost of a.
- For each neighbor b of a, it checks if the cost of b is less than min. If it is, it updates min to the cost of b.
- Finally, it sets the cost of a to min. This ensures that the cost of each vertex is the minimum cost of any vertex reachable from it.

```
procedure CHEAPEST-NODE-REACHABLE(G, costs)
    n ← |V|
    T[1...n] ← TOPOLOGICAL-SORT(G)
    visited[1...n] ← [FALSE, ..., FALSE]
    for a ∈ [n...1] do
        if not visited[a] then
            min ← costs[a]
            for neighbor b of T(a) do
                if min > costs[b] then
                    min ← costs[b]
                end if
            end for
            costs[a] ← min
            visited[a] ← TRUE
        end if
    end for
end procedure
```

- This algorithm works because in a topologically sorted DAG, every edge goes from a vertex earlier in the order to a vertex later in the order. Therefore, by the time the algorithm considers a vertex, it has already considered all vertices reachable from it, and so it can correctly compute the minimum cost.

Problem 3.25b. Give a linear-time algorithm that works for all directed graphs.

- To handle cycles in the graph, we can use SCCs.
- We can use Kosaraju's algorithm to find all SCCs in the graph. Then, for each SCC, we find the minimum cost vertex and assign this cost to the entire SCC
- Finally, we perform a similar process as in the DAG case, but now considering the SCCs instead of individual vertices.

```
procedure CHEAPEST-NODE-REACHABLE(G, costs)
    n ← |V|
    SCCs ← KOSARAJU(G)
    SCC_costs[1...n] ← [∞, ∞, ..., ∞]
    for each SCC in SCCs do
        min ← ∞
        for each vertex v in SCC do
            if costs[v] < min then
                min ← costs[v]
            end if
        end for
        for each vertex v in SCC do
            SCC_costs[v] ← min
        end for
    end for
    T[1..n] ← TOPOLOGICAL-SORT(SCCs)
    for a in [n...1] do
        min ← SCC_costs[a]
        for each neighbor b of T[a] do
            if min > SCC_costs[b] then
                min ← SCC_costs[b]
            end if
        end for
        SCC_costs[a] ← min
    end for
end procedure
```

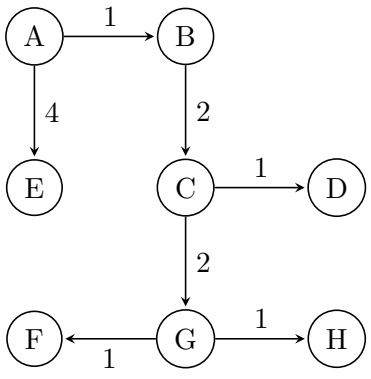
Run-time Analysis The run-time of this algorithm is $O(|V|+|E|)$ because Kosaraju's algorithm runs in $O(|V| + |E|)$ time, and the rest of the algorithm runs in $O(|V| + |E|)$ time.

Explanation of Correctness In this algorithm, we first find all the SCCs in the graph using Kosaraju's algorithm. We then find the minimum cost vertex in each SCC and assign this cost to the entire SCC. Finally, we perform a similar process as in the DAG case, but now considering the SCCs instead of individual vertices.

Problem 4.1a. Suppose Dijkstra's algorithm is run on the following graph, starting at node A. Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm.

$$\begin{bmatrix} A & B & C & D & E & F & G & H \\ 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 0 & 1 & \infty & \infty & 4 & 8 & \infty & \infty \\ 0 & 1 & 3 & \infty & 4 & 7 & 7 & \infty \\ 0 & 1 & 3 & 4 & 4 & 7 & 5 & \infty \\ 0 & 1 & 3 & 4 & 4 & 7 & 5 & 8 \\ 0 & 1 & 3 & 4 & 4 & 7 & 5 & 8 \\ 0 & 1 & 3 & 4 & 4 & 6 & 5 & 6 \end{bmatrix}$$

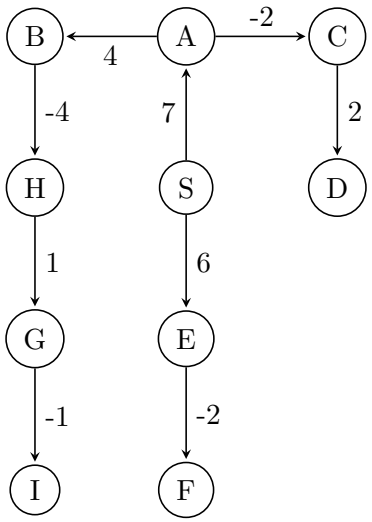
Problem 4.1b. Suppose Dijkstra's algorithm is run on the following graph, starting at node A. Show the final shortest-path tree.



Problem 4.2a. Suppose Bellman-Ford's algorithm is run on the following graph, starting at node S (found from topologically sorting). Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm.

$$\begin{bmatrix} S & A & B & C & D & E & F & G & H & I \\ 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 0 & 7 & \infty & 6 & \infty & 6 & 5 & \infty & \infty & \infty \\ 0 & 7 & 11 & 5 & 7 & 6 & 4 & \infty & 9 & \infty \\ 0 & 7 & 11 & 5 & 7 & 6 & 4 & 8 & 7 & \infty \\ 0 & 7 & 11 & 5 & 7 & 6 & 4 & 8 & 7 & 7 \end{bmatrix}$$

Problem 4.2b. Suppose Bellman-Ford's algorithm is run on the following graph, starting at node A. Show the final shortest-path tree.



Problem 4.5. Create a linear-time algorithm that returns the number of distinct shortest paths from u to v.

procedure COUNT-SHORTEST-PATHS (G, u, v)
shortest_distance ← BFS(G, u, v)
visited[1...n] ← [false, false, ..., false]
path_count ← 0
DFS(G, v, u, shortest_distance, 0, visited, path_count)
return path_count

procedure DFS $(G, current, target, sd, cd, visited, pc)$
if cd > sd **then**
    return
end if
if current = target and cd = sd **then**
    pc ← pc + 1
    return
end if
visited[current] ← true
for each neighbor of current **do**
    if not visited[neighbor] **then**
        DFS(G, neighbor, target, sd, cd + 1, visited, pc)
    end if
end for
visited[current] ← false

Run-time Analysis The run-time of this algorithm is $O(|V|+|E|)$ because BFS and DFS run in $O(|V| + |E|)$ time. The modified DFS runs in $O(|V| + |E|)$ time because it visits each vertex and edge at most once due to the visited array.

Explanation of Correctness In this algorithm, BFS is a function that runs Breadth-First Search on the graph G from vertex u to vertex v and returns the shortest distance. DFS is a function that performs Depth-First Search on the graph G from the current vertex to the target vertex, keeping track of the current distance and the shortest distance, and increments *path_count* whenever it finds a path from v to u with the same length as the shortest path.

Problem 344helper. Design a BFS-based algorithm to determine if a given undirected graph is bipartite. A graph is bipartite if its vertices can be divided into two disjoint sets such that every edge connects a vertex in one set with a vertex in the other set.

- The algorithm begins by initializing an array of colors for each vertex. The colors are 0 and 1.
- It then performs a BFS on the graph, starting at an arbitrary vertex.
- For each vertex, it assigns the opposite color of its parent to the vertex. If the vertex already has a color, it checks if the color is the same as the parent's color. If it is, then the graph is not bipartite.
- If the algorithm completes without finding a conflict, then the graph is bipartite.

Run-time Analysis The run-time of this algorithm is $O(|V|+|E|)$ because BFS runs in $O(|V| + |E|)$ time. The algorithm visits each vertex and edge at most once.

Explanation of Correctness In this algorithm, BFS is a function that runs Breadth-First Search on the graph G and assigns colors to each vertex. The algorithm then checks if the colors of the vertices are consistent with the definition of a bipartite graph.