More Graphs

1A

Pseudocode

```
Procedure LongestPathStartingAt(u, G)
        table // -1
        table[u] = 0

        incomingCount = CountIncomingEdges(G)
        corners = [vertices with 0 incoming edges from incomingCount]
        order = GetTopologicalOrder(corners, G)

        For v in order[u...]
                For out in v.outgoingNeighbors
                        table[out] = max(table[out], table[v] + 1)

        return table
```

Justification

The above algorithm works because of the optimal substructure in this problem and the idea of relaxation discussed in class. When vertex `v` is processed, it has been relaxed so that `table[v]` equals the longest path from `u` to `v`. It is true that `table[v]` is the longest path from `u` to `v` because of the optimal substructure of this problem. The value of `table[v]` depends on the value stored in `table` for all its parent vertices. All of `v`'s parent vertices had their longest path computed. Therefore, `table[v]` must equal the longest path from `u`; if not, I would have computed a different value for one of `v`'s parents.

Analysis

The algorithm counts incoming edges for every vertex $O(V+E)$; gets all the corner vertices $O(V)$; computes the topological order of `G` $O(V+E)$; iterates over every vertex and edge to compute the longest path $O(V+E)$. Overall $O(V+E)$

1B

Pseudocode

```
Procedure LongestPathStartingAt(G)
        incomingCount = CountIncomingEdges(G)
        corners = [vertices with 0 incoming edges from incomingCount]
        order = GetTopologicalOrder(corners, G)

        table // -1s
        // not all vertices may be reachable by any other vertex
        table[for all vertices with 0 incoming edges from incomingCount] = 0

        For v in order
                For out in G[v]
                        table[out] = max(table[out], table[v] + 1)

        return max(table)
```

Justification

The justification for this algorithm is similar to `#1A`. The difference is that there is no starting vertex. To find the longest path in this DAG, I must start at an edge that is not reachable by any other. If I did not, I could go one edge backward and find a path that is longer by one. Therefore, the corner edges are all potential starting points for the longest path and thus have a value of 0 in the table. Then, `table[v]` for any `v` is relative to some corner edge.

Analysis

The algorithm counts incoming edges for every vertex $O(V + E)$; gets all the corner vertices $O(V)$; computes the topological order of `G` $O(V + E)$; sets `table[v] = 0` for all `v` with zero incoming edges $O(V)$; iterates over every vertex and edge to compute the longest path $O(V + E)$; grabs the edge with the maximum value in order $O(V)$. Overall $O(V + E)$

2

Pseudocode

```
IsHamiltonian(G)
        incomingCount = CountIncomingEdges(G)
        corners = [vertices with 0 incoming edges from incomingCount]
        order = GetTopologicalOrder(corners, G)

        // checking if every vertex reaches the immediately subsequent
        For i in range [0, length of G-1)
                If order[i+1] not in G[i].outgoingNeighbors
                        Return False

        Return True
```

Justification

The above algorithm works because a Hamiltonian DAG (HDAG) has a topological order where every vertex leads immediately to the next. Such a topological order occurs iff there is precisely one vertex with 0 incoming edges (corner vertex) at every stage of building the order. This statement on topological order is true because if, at any stage, there is more than one corner vertex, the next vertex in the order will not be the child of any of the previous vertices, thus not being immediately connected with the previous vertex. The presence of a vertex that is not immediately connected with the previous vertex indicates that the graph is not an HDAG because there will be no single continuous path that can be taken to visit all vertices. Thus, building a topological order and checking if every node leads to the immediate subsequent is enough to determine if a DAG is an HDAG.

Analysis

The algorithm counts incoming edges for every vertex $O(V+E)$; gets all the corner vertices $O(V)$; computes the topological order of `G` $O(V+E)$; iterates over every vertex to see if the vertices in the topological order lead immediately to the next $O(V)$. Overall $O(V+E)$

#homework