Instructor Name: Siddhartha Nath
Email: s.nath@northeastern.edu

Northeastern University

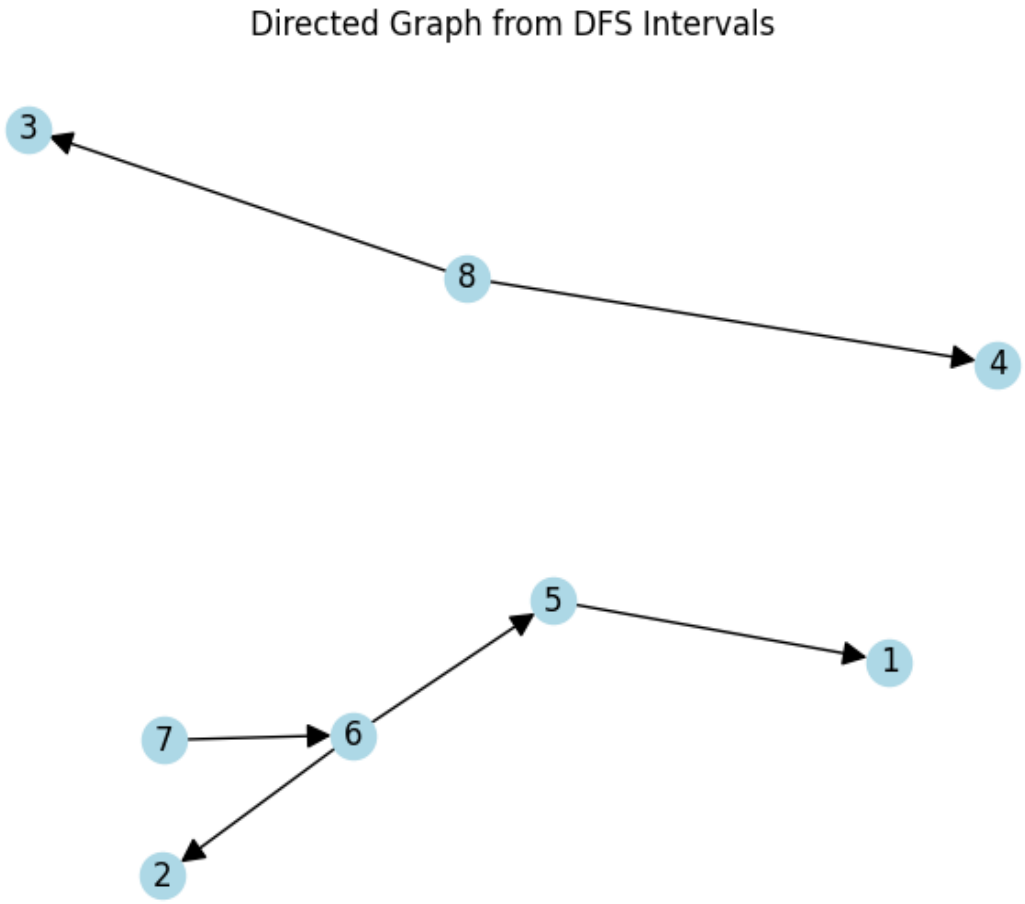
Program Structure and Algorithms (INFO 6205)
Homework #3 – 100 points

Student NAME: Xijing Zhang

Student ID: 002205911

Question 1 (15 points). *Suppose you are given a directed graph* $G = (V, E)$ *with* $V = \{1, 2, 3, 4, 5, 7, 8\}$ *and the depth first intervals* $([pre, post])$ *of each vertex are as follows.* $\{1 : [4, 5], 2 : [7, 8], 3 : [12, 13], 4 : [14, 15], 5 : [3, 6], 6 : [2, 9], 7 : [1, 10], 8 : [11, 16]\}$.

(a) (7 points) Draw this directed graph using networkx package in Python and include the image.



(b) *(3 points) What are the descendent and ancestor vertices of vertex 6?*

Descendants: 5, 1, 2

Ancestors: 7

(c) *(2 points) How many connected components does the graph have?*

The graph has 2 connected components.

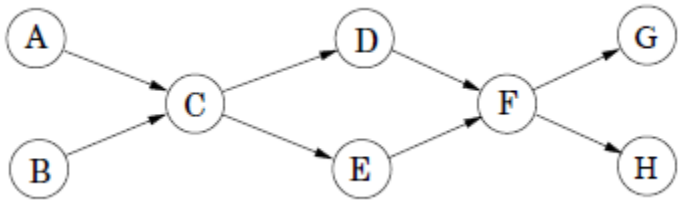
Component 1: Vertices 7, 6, 5, 1, and 2

Component 2: Vertices 8, 3, and 4

(d) *(3 points) Identify three pairs of vertices that form a cross edge (i.e., one is neither a descendent nor an ancestor of the other).*

(1, 2), (7, 8), (3, 4)

Question 2 (20 points)**.** *Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that appears first in alphabetical order.*



(a) *(12 points) Indicate the pre- and post- numbers of the nodes.*

A: Pre(A) = 1, Post(A) = 14

B: Pre(B) = 15, Post(B) = 16

C: Pre(C) = 2, Post(C) = 13

D: Pre(D) = 3, Post(D) = 10

E: Pre(E) = 11, Post(E) = 12

F: Pre(F) = 4, Post(F) = 9

G: Pre(G) = 5, Post(G) = 6

H: Pre(H) = 7, Post(H) = 8

(b) *(4 points) What are the sources and sinks of the graph?*

Sources: A,B

Sinks: G,H

(c) *(2 points) Write down one topological ordering found by the algorithm.*

B, A, C, E, D, F, H, G

(d) *(2 points) How many topological orderings does this graph have?*

A and B can appear in 2 different orders.

Then C must come next (only 1 way).

Then D and E can appear in 2 orders.

Then F must come next (1 way).

Finally, G and H can appear in 2 orders.

Multiplying $2 \times 1 \times 2 \times 1 \times 2 = 8$.

Question 3 (15 points). *Given an example of a graph with n vertices for which the queue of Breadth-first Search (BFS) will have $n-1$ vertices at one time, whereas the height of the recursion tree of Depth-First Search (DFS) is at most one. Both searches are started from the same vertex.*

BFS Analysis:
Start from vertex 1. All the other vertices $\{2, 3, \ldots, n\}$ are directly connected to vertex 1 and are added to the BFS queue immediately. Thus, the queue will have $n-1$ vertices at one time.

DFS Analysis:
Start from vertex 1. DFS visits one neighbor (say vertex 2), then backtracks to 1 to visit the next neighbor, and so on. Since each neighbor is directly connected to 1 and has no further neighbors, the recursion tree has a height of at most 1.

Question 4 (15 points). *You are given a binary tree $T = (V, E)$ that is not skewed and $|V| \geq 2$. Please describe in English a linear (in terms of $|V|$ and $|E|$) time algorithm to find the maximum sum of a path between any two leaves in T. Please explain why you are algorithm running time in linear.*

```
function findMaxPath(node):
    if node is a leaf:
        return node.value

    leftSum  = -
    rightSum = -

    if node.left exists:
        leftSum = findMaxPath(node.left)
    if node.right exists:
        rightSum = findMaxPath(node.right)

    if node.left exists and node.right exists:
        // Update global maximum with path through this node
        globalMax = max(globalMax, leftSum + node.value + rightSum)
        // Return max path sum from this node down to a leaf
        return node.value + max(leftSum, rightSum)
    else:
        // Only one child exists, return path through that child
        return node.value + (leftSum if node.left exists else rightSum)

------------------------------------------------
Initialize globalMax to -
Call findMaxPath(root)
The answer is in globalMax
```

Why It's Linear:
Each node is visited only once.
We do a few simple calculations per node.
Total time is O(V), which is linear.

Question 5 (35 points). *For each vertex u in an undirected graph, let twodegree[u] be the sum of the degrees of u's neighbors. You are given an undirected graph $G = (V, E)$ in adjacency-list format.*

(a) *(15 points) Please describe in English an efficient algorithm to compute the entire array of twodegree[·] values in time linear in $|V|$ and $|E|$.*

First, we precompute the degree of each vertex. For every vertex u, we determine its degree by finding the length of its adjacency list. Since the graph is undirected, every edge appears in the lists of both of its endpoints, so the total work for this step is proportional to the sum of the lengths of all adjacency lists, that is $O(|V| + |E|)$.

Next, we compute the twodegree value for each vertex u. We initialize twodegree[u] to zero, then iterate through every neighbor v of u (using u's adjacency list) and add the degree of v (computed in the previous step) to twodegree[u]. Again, since the total number of neighbor iterations over all vertices is proportional to the number of edges, this part also takes $O(|E|)$ time.

(b) *(15 points) Please write the pseudocode of your algorithm in (a).*

```
Algorithm ComputeTwoDegree(Graph G)
    Input: Graph G = (V, E) in adjacency-list format
    Output: Array twodegree[·] where for each u  V, twodegree[u] is the sum of degrees of u's

    // Step 1: Compute degree of each vertex
    for each vertex u in V do
        degree[u] ← length(Adj[u])

    // Step 2: Compute twodegree for each vertex
    for each vertex u in V do
        twodegree[u] ← 0
        for each vertex v in Adj[u] do
            twodegree[u] ← twodegree[u] + degree[v]

    return twodegree
```

(c) *(5 points) Please explain why the running time of your pseudocode in (b) is linear in $|V|$ and $|E|$.*

The running time is linear because the first step of computing the degrees takes time proportional to the sum of the lengths of the adjacency lists (i.e., $O(|V| + |E|)$). The second step, which iterates over each vertex and then over each neighbor in its list, similarly takes time proportional to $O(|E|)$. Thus, the overall running time is $O(|V| + |E|)$, which is linear in the size of the graph.