1. [12 points] Consider the DirectedDFS algorithm 13.11 from the book or as given in class. It does not distinguish between forward and cross edges.

   a. Modify this algorithm to differentiate between these edges; that is, give its pseudocode. Thus, edges can be one of four types: tree, back, cross or forward.

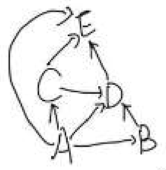
   Algorithm ModifiedDirectedDFS(G, v):

```
Label v as active    // Every vertex is initially unexplored
for each outgoing edge, e, that is incident to v in G do
    if e is unexplored then
        Let w be the destination vertex for e
        if w is unexplored and not active then
            Label e as a discovery edge
            ModifiedDirectedDFS(G, w)
        if w is active then
            Label e as a back edge
```

   // Second for-loop to distinguish forward and cross edges

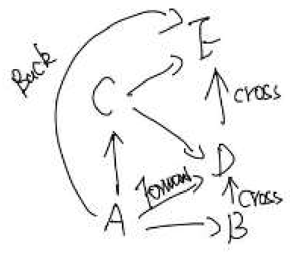
```
for each incoming edge, e = (w, v)
    if w is active and e is unexplored then
        Label e as a forward edge
    else if e is unexplored then
        Label e as a cross edge
Label v as explored
```

b. Give an example of running your algorithm.



MDDFS(G,A)
A→C discovery edge
MDDFS(G,C)
C→E discovery edge
MDDFS(G,E)
A→E back edge
D→E cross edge
E is explored
back to MDDFS(G,C)
C→D discovery
MDDFS(G,D)
A→D Forward edge
B→D Cross edge
D is explored
back to MDDFS(G,C)
C is explored

back to MDDFS(G,A)
A→B discovery edge
MDDFS(G,B)
B is explored
back to MDDFS(G,A)
A is explored



c. Prove its correctness for labeling these four edges and its running time.

Correctness Proof:

The algorithm correctly categorizes edges based on their nature:

Discovery Edges: The algorithm labels an edge as a discovery edge when it visits an unexplored vertex (w) from an active vertex (v). This ensures that we are exploring new parts of the graph, and this categorization is accurate.

Back Edges: The algorithm labels an edge as a back edge when it encounters a vertex (w) that is already labeled as "active." This indicates a cycle in the graph, and marking the edge as a back edge is correct.

Forward Edges: The algorithm labels an edge as a forward edge when it explores a vertex (w) that is still active, and the edge is unexplored. This indicates that it's connecting to a vertex in the same DFS branch, so it's a forward edge.

Cross Edges: The algorithm labels an edge as a cross edge when it explores a vertex (w) that is not active, and the edge is unexplored. This indicates that the edge connects to a different branch of the DFS tree, making it a cross edge.

Running Time:

The running time of the ModifiedDirectedDFS algorithm depends on the number of vertices (V) and edges (E) in the graph, which is O(V + E).

In the worst case, each edge and vertex are visited once during the DFS traversal, resulting in O(V + E) time complexity.

2. [10 points, 4 points for a and 6 points for b]

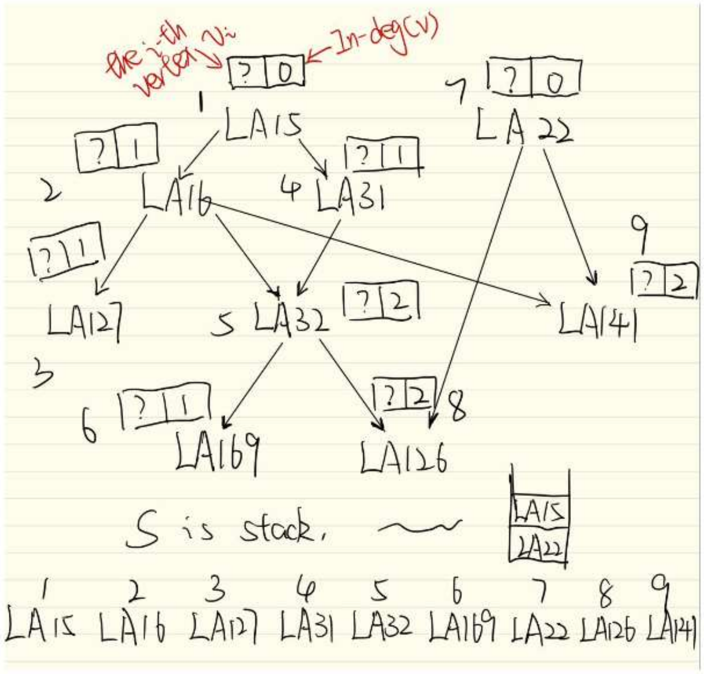
Bob loves foreign languages and wants to plan his course schedule to take the following nine language courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA169. The course prerequisites are:

- LA15: (none)
- LA16: LA15
- LA22: (none)
- LA31: LA15
- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32.

a. Draw a directed graph to model the course and prerequisites.

$$\begin{tikzcd}
 & & \text{LA15} \arrow[dl] \arrow[dr] & & \text{LA22} \arrow[ddr] \arrow[dddl] & \\
 & \text{LA16} \arrow[dl] \arrow[dr] \arrow[drrrr] & & \text{LA31} \arrow[dl] & & \\
\text{LA127} & & \text{LA32} \arrow[dl] \arrow[dr] & & & \text{LA141} \\
 & \text{LA169} & & \text{LA126} & &
\end{tikzcd}$$

b. Find the sequence of courses that allows Bob to satisfy all the prerequisites using the topological sort algorithm explained in class. Consider the prerequisites as an adjacency list in the order that prerequisites are given. Thus, there will be a distinct topological sort ordering based on the adjacency list. Example, for LA32, the adjacency list is LA16, LA31. LA16 is first in the list followed by LA31.

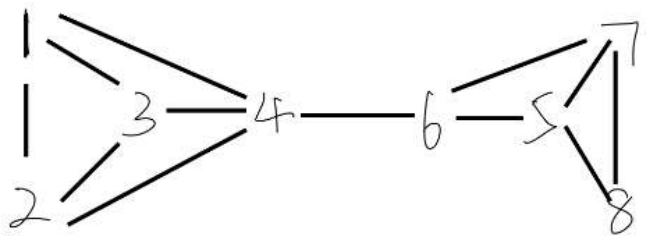


3. [22 points, 4 points for a, 4 points for b, 7 points for c, 7 points for d]

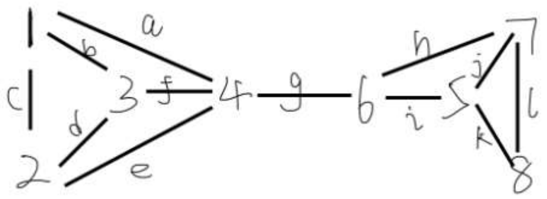
Let G be a graph whose vertices are the integers 1 through 8, and let the adjacent vertices of each vertex be given by the table below:

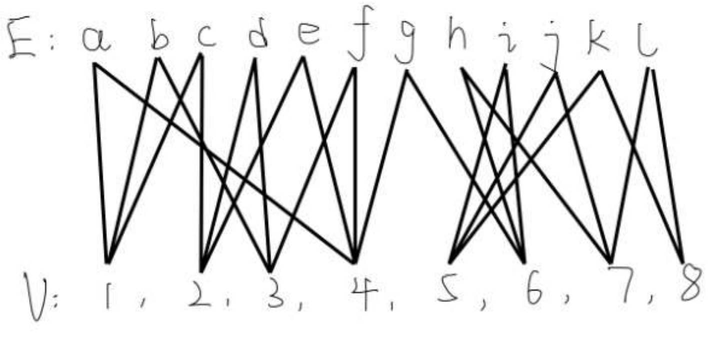
vertex	adjacent vertices
1	(2, 3, 4)
2	(1, 3, 4)
3	(1, 2, 4)
4	(1, 2, 3, 6)
5	(6, 7, 8)
6	(4, 5, 7)
7	(5, 6, 8)
8	(5, 7)

a. Draw G.

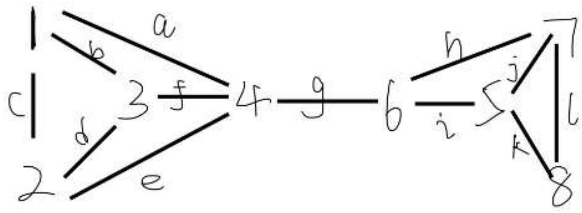


b. Draw a schematic representation of the adjacency list structure of G like Figure 13.3.

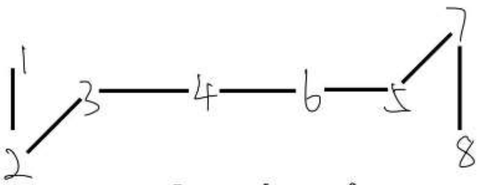




c. Order the vertices as they are visited in a DFS traversal starting at vertex 1. Also, draw the spanning tree that represents this.



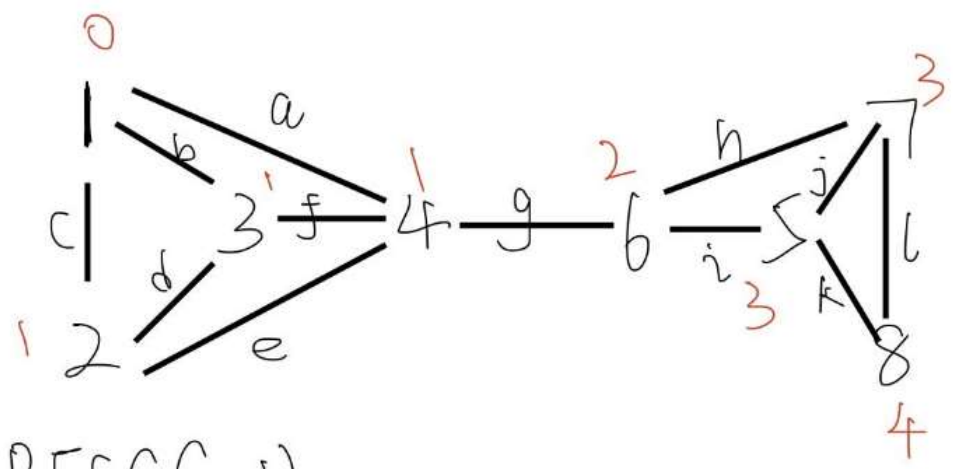
DFS(G,1)
DFS(G,2)
DFS(G,3)
DFS(G,4)
DFS(G,6)
DFS(G,5)
DFS(G,7)
DFS(G,8)



spanning tree

Order(1,2,3,4,6,5,7,8)

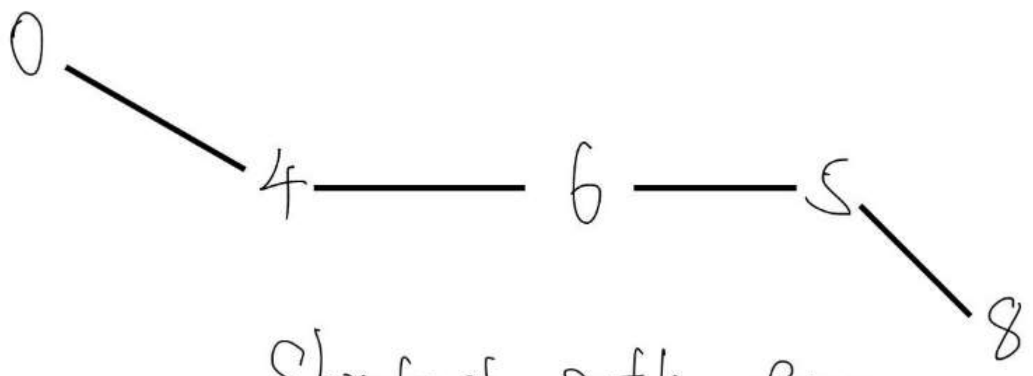
d. Order the vertices as they are visited in a BFS traversal starting at vertex 1. Also, draw the shortest path tree that represents this.



BFS(G,1)

order:

1 2 3 4 6 5 7 8



Shortest path. tree

4. [30 points – 10 points for each part a, b and c] Justify Theorem 13.11. You will need to prove each part a, b and c, of the theorem. Consider using induction or a contradiction.

Theorem 13.11: Let G be an undirected graph with n vertices and m edges. Then we have the following:

a. If G is connected, then $m \geq n - 1$

To justify Theorem 13.11(a) using mathematical induction, we'll use the principle of induction to prove that for any connected graph with n vertices, the number of edges m must be greater than or equal to n - 1.

Base Case (n = 1):

For the base case, consider a connected graph with a single vertex (n = 1). In this case, there are no edges (m = 0). Since 0 is greater than or equal to 1 - 1, the base case holds.

Inductive Hypothesis:

Assume that the theorem is true for some positive integer k, where k ≥ 1. That is, for any connected graph with k vertices, m ≥ k - 1.

Inductive Step:

We want to prove the theorem for k + 1 vertices. Consider a connected graph G with k + 1 vertices.

Let's choose one vertex v from the graph G. Removing this vertex along with all its incident edges will result in a subgraph G' with k vertices. By the inductive hypothesis, we know that G' must have at least k - 1 edges: m' ≥ (k - 1).

Now, add vertex v back to G along with its incident edges. Since G was connected, when we add vertex v back, it connects to at least one vertex in G' (otherwise, G would not be connected). Adding v creates at least one new edge in G.

Therefore, the number of edges in G, m, can be expressed as m = m' + 1 (for the new edge connecting v). Since m' ≥ (k - 1), we have m = m' + 1 ≥ (k - 1) + 1 = k.

This proves that for a connected graph with k + 1 vertices, the number of edges m is at least k, which is the same as k + 1 - 1. Therefore, the theorem holds for k + 1 vertices.

By the principle of mathematical induction, we have shown that the theorem is true for all positive integers n. This completes the proof of Theorem 13.11(a).

b. If G is a tree, then $m=n-1$

Base Case (n = 1):

For the base case, consider a tree with a single vertex (n = 1). In this case, there are no ed
ges (m = 0). Since 0 is equal to 1 - 1, the base case holds.

Inductive Hypothesis:

Assume that the theorem is true for some positive integer k, where k ≥ 1. That is, for any tre
e with k vertices, m = k - 1.

Inductive Step:

We want to prove the theorem for k + 1 vertices. Consider a tree T with k + 1 vertices.

Let's choose one leaf vertex v from the tree T. A leaf vertex is a vertex with only one edge.
When we remove this leaf vertex and its incident edge, we obtain a subtree T' with k vertices.
By the inductive hypothesis, we know that T' must have k - 1 edges: m' = k - 1.

Adding back the leaf vertex v along with its incident edge will result in the original tree T.
Adding v to T increases the number of edges by 1, so m = m' + 1.

Therefore, the number of edges in T, m, can be expressed as m = m' + 1 = (k - 1) + 1 = k.

This proves that for a tree with k + 1 vertices, the number of edges m is equal to k, which is
the same as k + 1 - 1. Therefore, the theorem holds for k + 1 vertices.

By the principle of mathematical induction, we have shown that the theorem is true for all pos
itive integers n.

c. If G is a forest, then $m \leq n - 1$

Base Case (n = 1):

For the base case, consider a forest with a single vertex (n = 1). In this case, there are no edges (m = 0). Since 0 is less than or equal to 1 - 1, the base case holds.

Inductive Hypothesis:

Assume that the theorem is true for some positive integer k, where k ≥ 1. That is, for any forest with k vertices, m ≤ k - 1.

Inductive Step:

We want to prove the theorem for k + 1 vertices. Consider a forest F with k + 1 vertices.

Let's choose one connected component of the forest F, which itself is a forest with some number of vertices and edges. By the inductive hypothesis, we know that the number of edges in this connected component is less than or equal to the number of vertices in that component minus 1.

Now, consider the remaining part of the forest F, which is also a forest with a certain number of vertices and edges. By the inductive hypothesis, the number of edges in this part is less than or equal to the number of vertices in this part minus 1.

Combining these two results, the total number of edges in the entire forest F is the sum of the edges in the two connected components, and it is less than or equal to the sum of the vertices in these components minus 1.

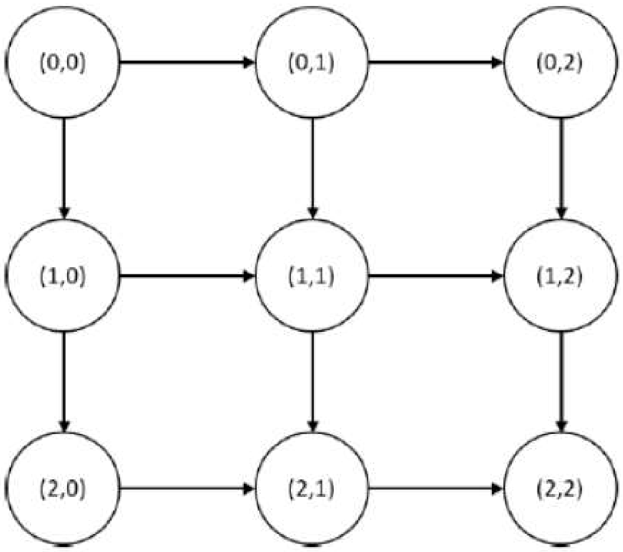
Since F has k + 1 vertices, this sum is k + 1, and we have shown that m ≤ (k - 1) + (k - 1) = 2k - 2. We want to show that m ≤ k.

It's easy to see that 2k - 2 is less than or equal to k when k is greater than or equal to 1. Therefore, m ≤ k.

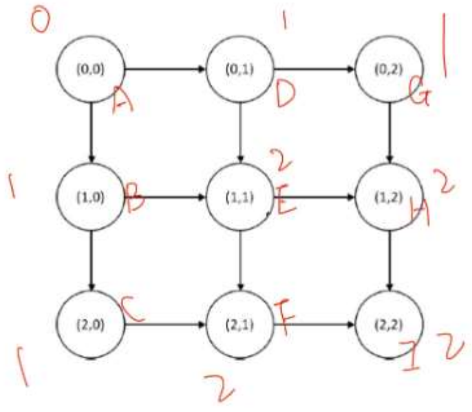
This proves that for a forest with k + 1 vertices, the number of edges m is less than or equal to k, which is the same as k + 1 - 1. Therefore, the theorem holds for k + 1 vertices.

By the principle of mathematical induction, we have shown that the theorem is true for all positive integers n.

5. [16 pts] This question pertains to topological sorting on a grid graph.



a. Give four distinct orderings of the nodes in the following directed acyclic graph.



Order 1: A B C D E F G H I

Order 2: A D G B E H C F I

Order 3: A B C D G E H F I

Order 4: A D G B C E F H I.

b. Generalize your four orderings to the case of the graph with the same connectivity pattern on an $n \times n$ grid.

STEP 1 ->First of all we will put (0*0)th node in ordering because The first vertex in topolog icalsorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).
STEP 2 -> secondly, we will put node(1*0)th or (0*1)th into our ordering.
STEP 3 ->repeatdly we will put vertex in our ordering as vertices such that for every directed edge (u,v), vertex u comes before v in the ordering.Till we have no vertex left.

6. [16 pts] Consider an undirected graph G=(V, E). Describe an O(n+m) time algorithm based on BFS that returns the path between 2 vertices of a tree.

a. Explain the main concept of your algorithm.

The main concept of the modified BFS algorithm, BFS_Find_Path, is to find a path between a source vertex s and a target vertex t in a tree (or any graph) using a Breadth-First Search approach. Here's how the algorithm works:

We start by initializing an empty queue Q to store the vertices we need to explore and a dictionary parent to keep track of the parent of each vertex. The parent dictionary will be used to reconstruct the path from t back to s when we find the target.

We enqueue the source vertex s into the queue and mark its parent as None since it has no parent.

We enter a loop that continues as long as there are vertices in the queue Q to explore.

Inside the loop, we dequeue a vertex called current from the queue.

If current is equal to the target vertex t, we have found the path. We reconstruct the path from t to s by following the parent links. We store the path in a list and return it.

If current is not equal to the target t, we examine its neighbors. For each neighbor w, if w has not been visited before (i.e., not in the parent dictionary), we mark current as the parent of w and enqueue w into the queue.

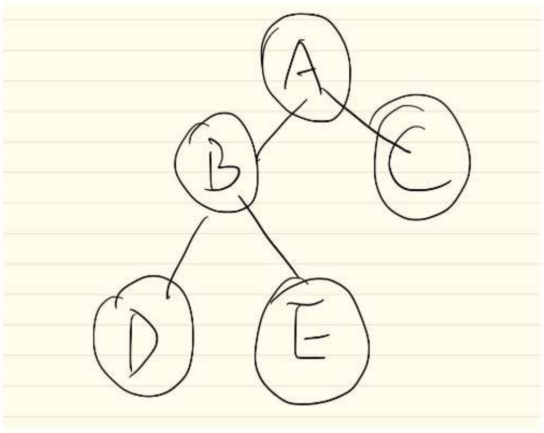
If we exhaust all possibilities in the loop and still haven't reached the target t, it means there is no path from s to t, so we return a message indicating that.

The key idea here is to use the BFS traversal to explore the graph systematically, keeping track of the parent of each vertex. When we reach the target vertex, we can backtrack through the parent links to reconstruct the path from t to s. This algorithm ensures that we find the shortest path between the two vertices in an unweighted graph (or tree) because BFS explores vertices in layers, guaranteeing that the first path found is the shortest.

b. Give pseudo-code.

```
Algorithm BFS_Find_Path(G, s, t):
    Input: A graph G, a source vertex s, and a target vertex t in G
    Output: A path from vertex s to vertex t in G, or an indication that no such path exists

    Create an empty queue, Q
    Create a dictionary, parent, to store the parent of each vertex
    Initialize parent[s] ← None

    Enqueue s into Q

    while Q is not empty do
        current ← Dequeue(Q)
        if current = t then
            # Reconstruct the path from t to s
            path ← []
            while current is not None:
                path.insert(0, current)
                current ← parent[current]
            return path
        for each neighbor, w, of current in G:
            if w is not in parent:
                parent[w] ← current
                Enqueue(Q, w)

    # No path from s to t was found
    return "No path exists"
```

c. Present an example of running your algorithm.



In this tree, vertex A is the root, and we want to find a path from vertex A to vertex E. We'll use the BFS_Find_Path algorithm for this.

1: Initialize the tree

G = { 'A': ['B', 'C'], 'B': ['D', 'E'], 'C': [], 'D': [], 'E': [] }

2: Call the BFS_Find_Path algorithm

path = BFS_Find_Path(G, 'A', 'E')

3: The algorithm will execute as follows

Start with Q = ['A'] and parent = {'A': None}.

Dequeue A, and explore its neighbors B and C. Enqueue B and set its parent as 'A'.

Dequeue B, explore its neighbors D and E, and enqueue D (parent is 'B') and E (parent is 'B').

Dequeue C, but it has no neighbors.

Dequeue D, but it has no neighbors.

Dequeue E, but it has no neighbors.

d. Prove/justify its correctness and its running time.

Correctness Proof:

The algorithm starts by enqueuing the source vertex s and setting its parent to None, which is a valid initial state.

The algorithm proceeds by exploring vertices in a breadth-first manner, which ensures that when we reach the target vertex t, we have found the shortest path from s to t. This is because BFS explores vertices at each level before moving on to the next level, and any path found during the traversal is guaranteed to be shorter than any subsequently found paths.

When the algorithm reaches the target vertex t, it constructs the path by backtracking through the parent dictionary, starting from t and following parent links until it reaches s. This ensures that the path is correctly constructed.

If the algorithm finishes without finding a path to t, it correctly returns a message indicating that no such path exists.

The algorithm guarantees that it finds a correct path (if it exists) and correctly reports when no path exists. Therefore, the algorithm is correct.

Running Time Analysis:

The running time of the BFS_Find_Path algorithm can be analyzed as follows:

Enqueuing and dequeuing each vertex once takes O(n), where n is the number of vertices in the graph.

In the worst case, the algorithm explores all edges, leading to O(m) edge examinations, where m is the number of edges in the graph.

Constructing the path by backtracking through the parent dictionary takes O(n) time in the worst case since it involves following the parent links from the target vertex back to the source.

Therefore, the overall time complexity of the algorithm is O(n + m), which is a linear time complexity with respect to the size of the input graph. This is an efficient way to find a path in a tree or any graph and ensures that the algorithm is suitable for practical applications even in large graphs.