Question 1 : Analysis of Directed Graph Properties

(a) A cross edge of a depth-first tree of a directed graph G can be contained in a simple directed cycle in G:

Example:

Consider a directed graph G with vertices A, B, and C, and edges $(A \to B), (B \to C)$, and $(C \to A)$. When traversing the graph in a depth-first search (DFS), if we encounter the edge $(A \to B)$ as a cross edge, it can be part of the simple directed cycle $(A \to B \to C \to A)$. In this case, the cross edge is contained in a simple directed cycle.

(b) A forward edge of a depth-first tree of a directed graph G can be contained in a simple directed cycle in G:

Example:

Consider a directed graph G with vertices A, B, and C, and edges $(A \to B), (B \to C)$, and $(C \to A)$. When traversing the graph in a depth-first search, if we encounter the edge $(A \to B)$ as a forward edge, it can be part of the simple directed cycle $(A \to B \to C \to A)$. In this case, the forward edge is contained in a simple directed cycle.

A more in depth explanation!

This document explores the possibility of finding simple directed cycles in a directed graph G that contain edges belonging to a depth-first tree (DFS) of G. We will consider two types of edges within the DFS:

- **Cross edges:** These connect two nodes that are not in an ancestor-descendant relationship within the DFS tree.
- **Forward edges:** These connect a node to one of its descendants in the DFS tree.

The goal is to determine whether, under any circumstances, such edges can be part of simple directed cycles in the original graph G.

Case 1: Cross Edges

We can demonstrate that **cross edges from a DFS tree can indeed be part of simple directed cycles** in G. Consider the following example graph:

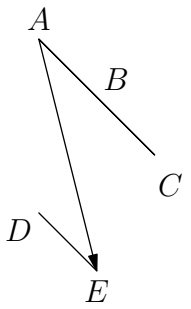


Figure 1: Example graph G with a cross edge in the DFS tree.

Performing a DFS starting from node A will result in the following tree:

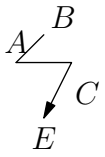


Figure 2: DFS tree of the example graph.

Here, edge AE is a cross edge, as neither A nor E are ancestor or descendant of each other in the tree. However, observe the cycle ABE. This cycle is simple (meaning it has no repeating nodes) and contains the cross edge AE. Therefore, a cross edge from the DFS tree can be part of a simple directed cycle in the original graph.

Case 2: Forward Edges

Forward edges present a slightly different scenario. By definition, no simple cycle within the DFS tree itself can contain a forward edge. This is because a cycle must return to the starting node and cannot "skip" levels in the tree hierarchy.

However, the situation changes when considering the original graph G. A forward edge in the DFS tree can become part of a simple cycle that goes **outside** the tree structure. In our example graph, if we perform the DFS again but choose to follow the edge AC instead of backtracking from C to B, we get the following tree:

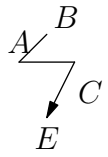


Figure 3: Alternative DFS tree with a forward edge.

In this case, edge BC is a forward edge, as B is a descendant of A in the tree. But consider the cycle ABC. This is a simple cycle in the original graph and it contains the forward edge BC.

Therefore, both **cross edges and forward edges from a DFS tree can be found in simple directed cycles in the original graph under certain conditions**. It depends on the specific structure of the graph and the chosen path during the DFS exploration.

Question 2 : Greedy Interval Covering Algorithm

Pseudocode

The greedy algorithm for finding the smallest set of closed intervals is as follows:

```
function greedy_interval_covering(A):
    n = length(A)
    intervals = []
    i = 0
    while i < n:
        current_interval = [A[i], A[i] + 1]
        while i + 1 < n and A[i + 1] == current_interval[1]:
            i = i + 1
            current_interval[1] = A[i] + 1
        if not intervals or current_interval[0] > intervals[-1][1]:
            intervals.append(current_interval)
        i = i + 1
    return intervals
```

Algorithmic Correctness

The correctness of the algorithm can be argued as follows:

1. The algorithm initializes an empty array `intervals` to store the closed intervals.

2. It iterates through the sorted array `A`, starting a new interval whenever it encounters a number not covered by the current interval.

3. The algorithm extends the current interval as long as the next element in `A` is strictly within its range.

4. It adds the current interval to the output only if it's not redundant, ensuring that the resulting set is minimal.

5. The algorithm continues this process until all elements in `A` are covered.

Linearity: $O(n)$

The time complexity of the algorithm is linear, denoted as $O(n)$, where n is the length of the input array `A`. Each element in the array is processed once, and the while loop iterates through the elements of `A` exactly once.

The operations inside the loop, such as comparisons and updates, take constant time. Therefore, the overall time complexity of the algorithm is $O(n)$, making it an efficient linear-time solution.

Full Code

Click here to open the file.

Dry Run

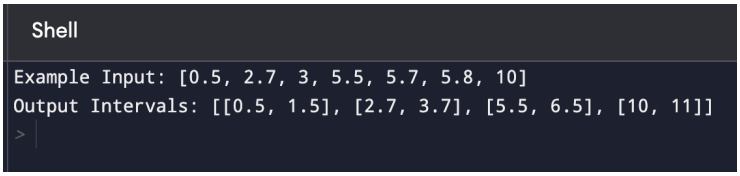


Figure 4: Greedy Interval Covering Problem

Question 3

Let us Generalize first with a numerical example!)

Graph:

$$V = \{(i, j) \in \mathbb{Z}^2 \mid 1 \leq i \leq n,\ 1 \leq j \leq m\}$$

$$E = \{(i, j), (i{+}1, j)\} \mid 1 \leq i < n,\ 1 \leq j \leq m\} \cup \{(i, j), (i, j{+}1)\} \mid 1 \leq i \leq n,\ 1 \leq j < m\}$$

Adjacency List Order:

The adjacency lists are ordered lexicographically, meaning that if (i, j) and (i', j') are vertices and $i < i'$ or $i = i'$ and $j < j'$, then $(i, j) < (i', j')$.

Breadth-First Tree (BFS)

Adjacency List:

```
(1, 1) -> (2, 1), (1, 2)
(2, 1) -> (3, 1), (2, 2)
(2, 2) -> (3, 2), (2, 3)
...
```

and so on...

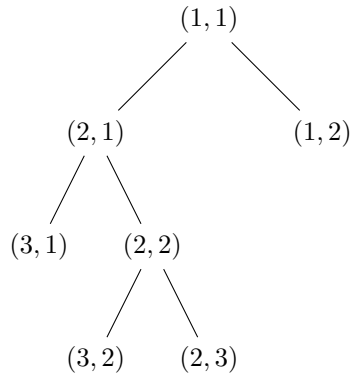
BFS Tree:

$$V' = V$$

$$E' = \{(i, m) \to (i + 1, m) \mid 1 \leq i < n\}$$

Explanation:

- The tree includes all vertices in the bottommost row ($1 \leq j \leq m$).
- Each vertex in the bottommost row is connected to its right neighbor.



Depth-First Tree (DFS)

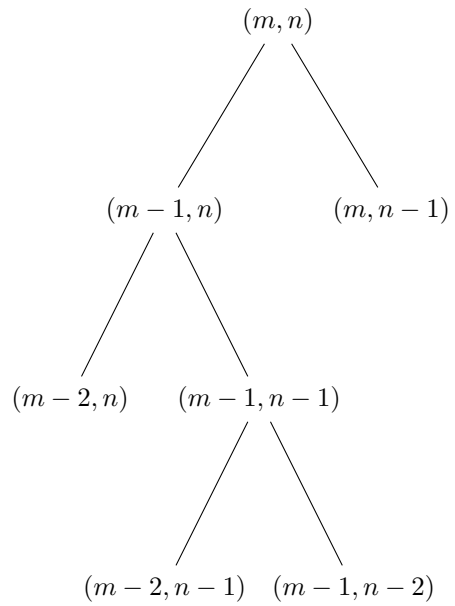
DFS Tree:

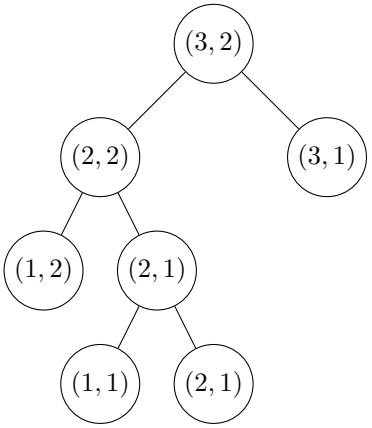
$$V' = V$$

$$E' = \{(n, j) \to (n, j+1) \mid 1 \leq j < m\}$$

Explanation:

- The tree includes all vertices in the rightmost column ($1 \leq i \leq n$).
- Each vertex in the rightmost column is connected to its upper neighbor. This means that the tree includes all vertices in the rightmost column, and each vertex in the rightmost column is connected to its upper neighbor.
- The crucial point in both descriptions is to include all vertices in either the bottommost row (for BFS) or the rightmost column (for DFS), and to connect each vertex in the chosen row or column to its neighboring vertex.





Additional Note: The order relation $(i, j) < (i', j')$ concerns only the list of adjacencies. During the search, vertices are explored based on the lexicographical order of their neighbors. For example, if $(m-1, n) < (m, n-1)$, during the search, $(m-1, n)$ is explored before $(m, n-1)$. The specified order of vertices in the adjacency lists determines the shape of the resulting tree.

Based on the below Adjacency List

Adjacency List:

$$(m, n) \to (m-1, n), (m, n-1)$$
$$(m-1, n) \to (m-2, n), (m-1, n-1)$$
$$(m-1, n-1) \to (m-2, n-1), (m-1, n-2)$$
$$\dots$$

NOW LET US APPLY THE TEXTBOOK DEF!

Breadth-First Search (BFS) with Root (1, 1)

Textbook Definitions

```
BFS(G, s)

for each vertex u belongs to G.V - {s}
    u.color = WHITE
    u.d = infinite
    u.pi = NIL
s.color = GRAY
s.d = 0
s.pi = NIL
Q = fi
```

```
ENQUEUE(Q, s)
while Q not equal to fi
    u = DEQUEUE(Q)
    for each vertex v in G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.pi = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Algorithm for our problem based on Textbook Definition!

We start with the vertex (1, 1) as the source vertex (s) and apply the BFS algorithm:

1. **Initialization:**
   - All vertices are initially set to WHITE.
   - Distance (d) is set to infinite, predecessor (pi) is set to NIL for each vertex.
   - The source vertex (1, 1) is colored GRAY, distance is set to 0, and predecessor is NIL.
   - The queue Q is initialized with the source vertex (1, 1).
2. **BFS Iteration:**
   - Dequeue the front vertex u from the queue.
   - For each vertex v in the adjacency list of u, if v is WHITE:
     - Color v GRAY.
     - Set distance (d) of v to $u.d + 1$.
     - Set predecessor (pi) of v to u.
     - Enqueue v into the queue.
   - Mark u as BLACK.
3. **Repeat:**
   - Continue this process until the queue is empty.

Result

The result is a breadth-first tree where the edges are explored level by level from the source (1, 1) vertex.

Resultant Breadth-First Tree

The breadth-first tree is formed by exploring vertices level by level from the source (1, 1).

Level 0: $(1, 1)$
Level 1: $(2, 1), (1, 2)$
Level 2: $(3, 1), (2, 2), (1, 3)$
...
Level $n-1$: $(n, 1), (n-1, 2), \ldots, (2, m-1), (1, m)$

Depth-First Search (DFS) with Root (n, m)

Textbook Definitions

```
DFS-Visit(G, u)

time = time + 1
u.d = time
u.color = GRAY
for each vertex v in G.Adj[u]
    if v.color == WHITE
        v.pi = u
        DFS-Visit(G, v)

time = time + 1
u.f = time
u.color = BLACK
```

Algorithm for our Solution based on Textbook Definition!

We start with the vertex (n, m) as the source vertex (u) and apply the DFS algorithm:

1. **DFS-Visit:**
   - Initialize time and set d for the source vertex to the current time.
   - Color the source vertex GRAY.
   - For each vertex v in the adjacency list of u, if v is WHITE:
     - Set predecessor (pi) of v to u.
     - Recursively call DFS-Visit for v.
2. **Backtracking:**
   - After visiting all neighbors, increment time and set f for the source vertex.
   - Mark the source vertex BLACK.

Result

The result is a depth-first tree where each vertex is explored as deeply as possible before backtracking. The edges are determined by the order of exploration based on the lexicographical order of the adjacency lists.

The depth-first tree is formed by exploring vertices as deeply as possible before backtracking from the source (n, m).

Source: (n, m)

Neighbors: $(n-1, m), (n, m-1)$

Further Exploration:

$(n-1, m) \Rightarrow (n-2, m), (n-1, m-1), (n-1, m+1)$

$(n, m-1) \Rightarrow (n, m-2), (n-1, m-1), (n+1, m-1)$

$(n-2, m) \Rightarrow (n-3, m), (n-2, m-1), (n-2, m+1)$

$\vdots$

$(2, m) \Rightarrow (1, m), (2, m-1), (2, m+1)$

$(1, m-1) \Rightarrow (1, m-2), (2, m-1), (1, m)$

$(1, m-2) \Rightarrow (1, m-3), (2, m-2), (1, m-1)$

$\vdots$

$(1, 1) \Rightarrow (1, 2), (2, 1)$

Question 4 : Graph's Vertex Importance

The algorithm is designed to find the importance of a vertex in an undirected connected graph. Given a graph $G = (V, E)$ represented as adjacency lists and a vertex $v \in V$ as input, the algorithm identifies the importance of the vertex based on the number of connected components resulting from the removal of the vertex and its incident edges.

Algorithm 1 Vertex Importance Calculation

```
1: procedure DFS(graph, visited, v, removed_vertex)
2:     visited.add(v)
3:     for neighbor in graph.get(v, []) do
4:         if neighbor not in visited and neighbor ≠ removed_vertex then
5:             DFS(graph, visited, neighbor, removed_vertex)
6: procedure CountComponents(graph, removed_vertex)
7:     visited ← set()
8:     components ← 0
9:     for vertex in graph do
10:        if vertex ≠ removed_vertex and vertex not in visited then
11:            DFS(graph, visited, vertex, removed_vertex)
12:            components ← components + 1
13:    return components
14: procedure PrintImportance(graph)
15:    for vertex in graph do
16:        importance ← CountComponents(graph, vertex)
17:        print("Vertex", vertex, " : Importance", importance)
```

Algorithm Correctness

The algorithm works by performing a depth-first search (DFS) on the graph for each vertex excluding the input vertex to be removed. The `CountComponents` function then counts the number of connected components formed after removing the specified vertex. This process is repeated for every vertex in the graph. The resulting importance values indicate how many connected components would be formed if each vertex were removed.

Explanation of Linearity

The time complexity of the algorithm is analyzed in terms of the size of the input, where V is the number of vertices and E is the number of edges in the graph. The DFS function visits each vertex and edge exactly once, resulting in a time complexity of $O(V + E)$.

Pseudo Code

```
# Provided code
def dfs(graph, visited, v, removed_vertex):
    visited.add(v)
    for neighbor in graph.get(v, []):
        if neighbor not in visited and neighbor != removed_vertex:
            dfs(graph, visited, neighbor, removed_vertex)

def count_components(graph, removed_vertex):
    visited = set()
    components = 0

    for vertex in graph:
        if vertex != removed_vertex and vertex not in visited:
            dfs(graph, visited, vertex, removed_vertex)
            components += 1

    return components

def print_importance(graph):
    for vertex in graph:
        importance = count_components(graph, vertex)
        print(f"Vertex {vertex} : Importance {importance}")

# Example usage
graph = {
    1: [2, 3, 4],
    2: [1, 3, 4],
    3: [1, 2, 4],
    4: [3, 2, 1]
}

print_importance(graph)
```

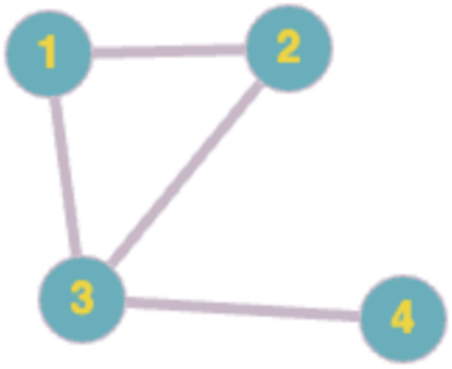
Here is the link to the full code :
Link

Running Tests for Different Conditions

Test Case 1:

```
graph = {
   1: [2, 3],
   2: [1,3],
   3: [1,2,4],
   4: [3]
}
```

Graph:



Output:

```
Shell
Vertex 1 : Importance 1
Vertex 2 : Importance 1
Vertex 3 : Importance 2
Vertex 4 : Importance 1
>
```

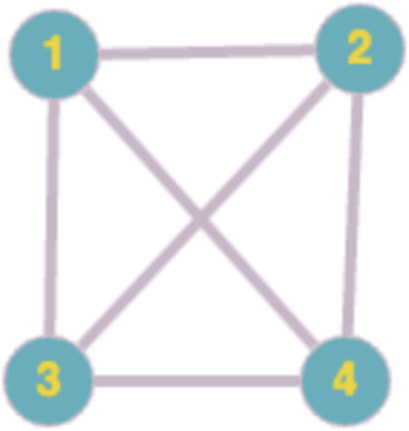
Test Case 2:

graph = {
1: [2, 3,4],
2: [1],
3: [1],
4: [1]

}

```
Shell
Vertex 1 : Importance 3
Vertex 2 : Importance 1
Vertex 3 : Importance 1
Vertex 4 : Importance 1
>
```

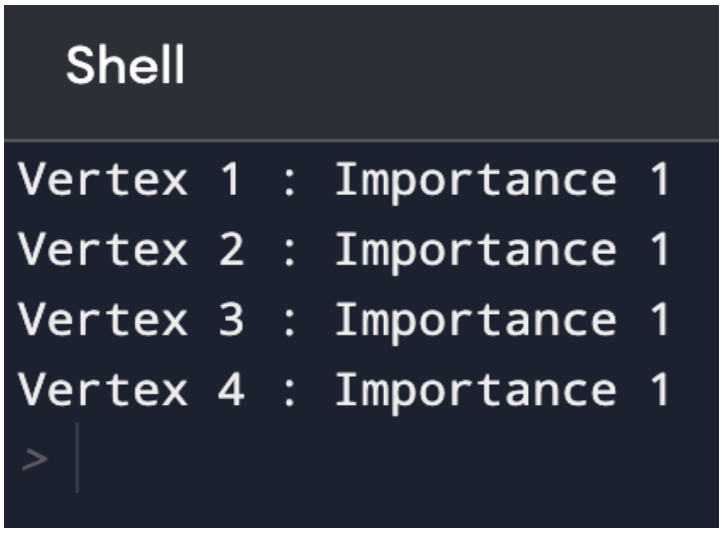
Test Case 3:




```
graph = {
   1: [2,3,4],
   2: [3,4,1],
   3: [1,2,4],
   4: [1,2,3]
```

}

Output


Shell
Vertex 1 : Importance 1
Vertex 2 : Importance 1
Vertex 3 : Importance 1
Vertex 4 : Importance 1
>