Assignment 10

Manan Patel

1. Efficiency Comparison of Adjacency Matrix and Adjacency List

Adjacency List

- **Advantages:**
  - Efficient for sparse graphs as it only stores information about existing edges.
  - Adding or removing edges is simpler and requires fewer updates since only specific edge entries are modified.
  - Space complexity is $O(V + E)$, where V is the number of vertices and E is the number of edges.
- **Disadvantages:**
  - In dense graphs, it may take more space due to the need to store pointers for multiple edges.
  - Traversal might require more time as accessing edges involves iterating through lists.

Adjacency Matrix

- **Advantages:**
  - Performs better in dense graphs due to its constant time ($O(1)$) edge access.
  - Space complexity is $O(V^2)$, independent of the number of edges, which is ideal for dense graphs.
- **Disadvantages:**
  - Inefficient for sparse graphs as it requires $O(V^2)$ space even when many entries are unused.
  - Adding or removing edges involves updating the entire matrix, which can be slower.

Conclusion

For sparse graphs, adjacency lists are generally more efficient due to their space-saving nature and ease of modification. For dense graphs, adjacency matrices perform better in terms of access time and storage efficiency.

2. Most Efficient Algorithm for the Problem

Solution A: Revised DFS for Each Vertex

Pseudocode:

```
function DFS(graph, u, visited, result):
    visited[u] = True
    result[u] = u  # Initialize with the current vertex

    for each v in graph[u]:  # Iterate over all neighbors
        if not visited[v]:
            DFS(graph, v, visited, result)
        result[u] = min(result[u], result[v])  # Update with the smallest re

function findSmallestReachable(graph):
    visited = [False] * number_of_vertices(graph)
    result = {}  # Store the smallest reachable vertex for each vertex

    for each vertex in graph:
        if not visited[vertex]:
            DFS(graph, vertex, visited, result)

    return result
```

Solution B: Using Transposed Graph

Pseudocode:

```
function transposeGraph(graph):
    transposed = create_empty_graph(number_of_vertices(graph))
    for u in graph:
        for v in graph[u]:
            add_edge(transposed, v, u)  # Reverse the edge direction
    return transposed

function DFS(graph, u, visited, label, result):
    visited[u] = True
    result[u] = label  # Mark with the smallest reachable label

    for each v in graph[u]:  # Traverse the neighbors
        if not visited[v]:
            DFS(graph, v, visited, label, result)

function findSmallestReachable(graph):
    transposed = transposeGraph(graph)
    visited = [False] * number_of_vertices(transposed)
    result = {}
    label = 1  # Start labeling with 1

    for vertex in range(number_of_vertices(transposed)):
        if not visited[vertex]:
```

```
        DFS(transposed, vertex, visited, label, result)
        label += 1

# Map all vertices in the same group to the smallest labeled vertex
smallest_labels = {}
for u, lbl in result.items():
    if lbl not in smallest_labels or u < smallest_labels[lbl]:
        smallest_labels[lbl] = u

for u in result:
    result[u] = smallest_labels[result[u]]

return result
```

3. Input and Output

Input: Adjacency List

```
1 -> 2
1 -> 4
2 -> 5
3 -> 6
3 -> 5
4 -> 2
5 -> 4
6 -> 6
```

Output: Adjacency List

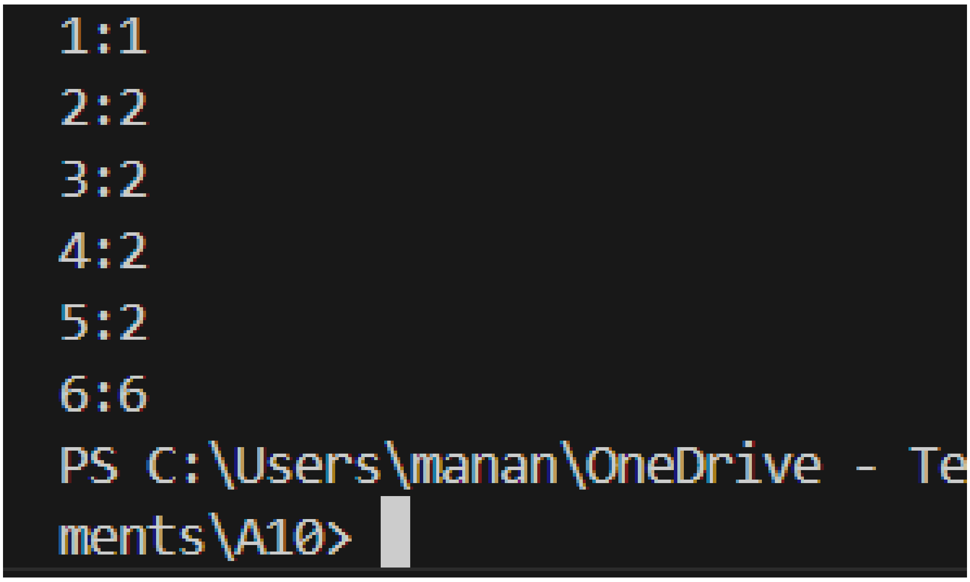

Figure 1: Output for Adjacency List

Input: Adjacency Matrix

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Output: Adjacency Matrix

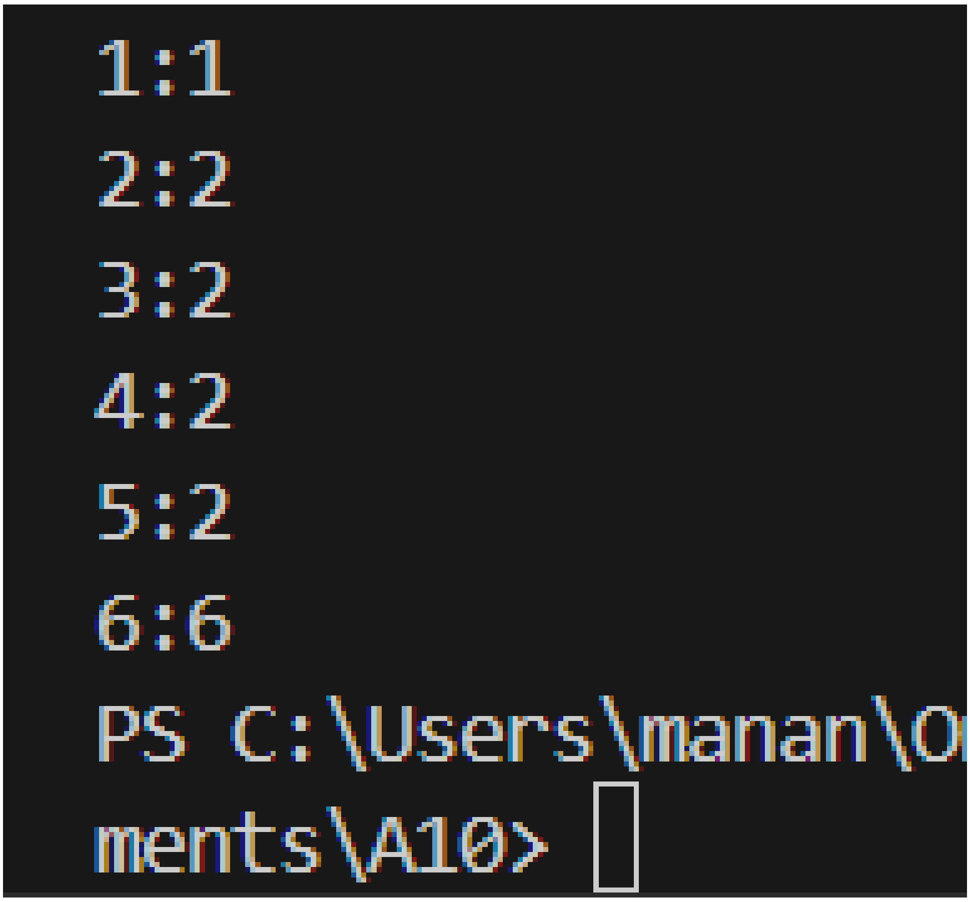

Figure 2: Output for Adjacency Matrix