CMPSC 413 - Lab 6 (BFS & DFS)

Graph Traversals Implementations

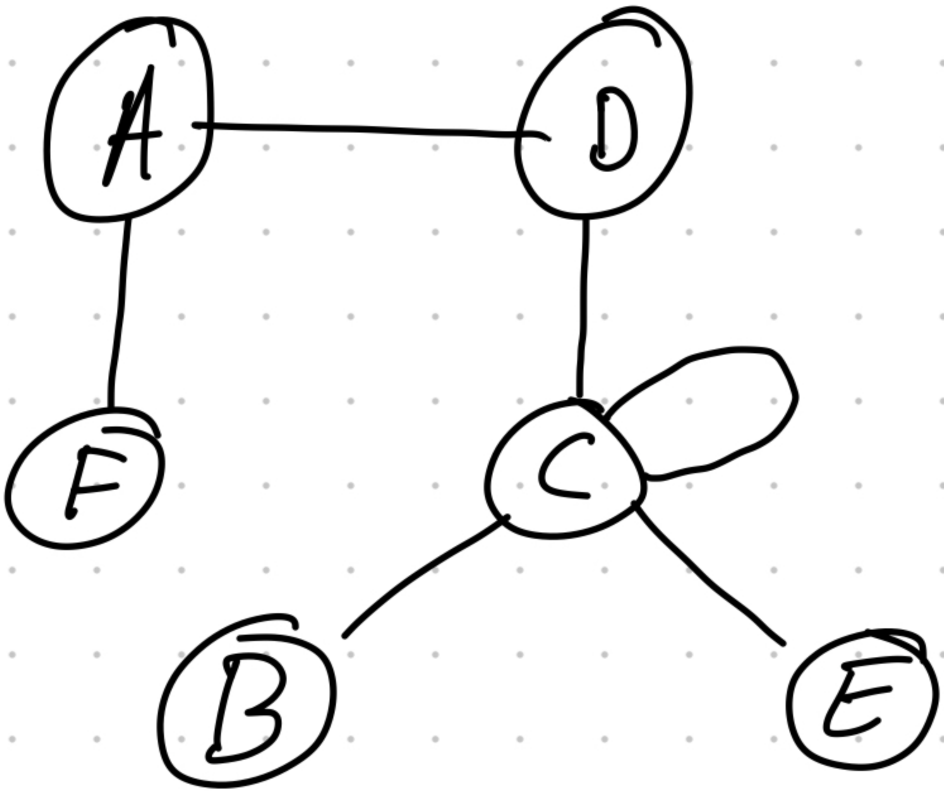
Lab Exercises:

1. Draw a sketch of below graphs and attach the screenshot here.

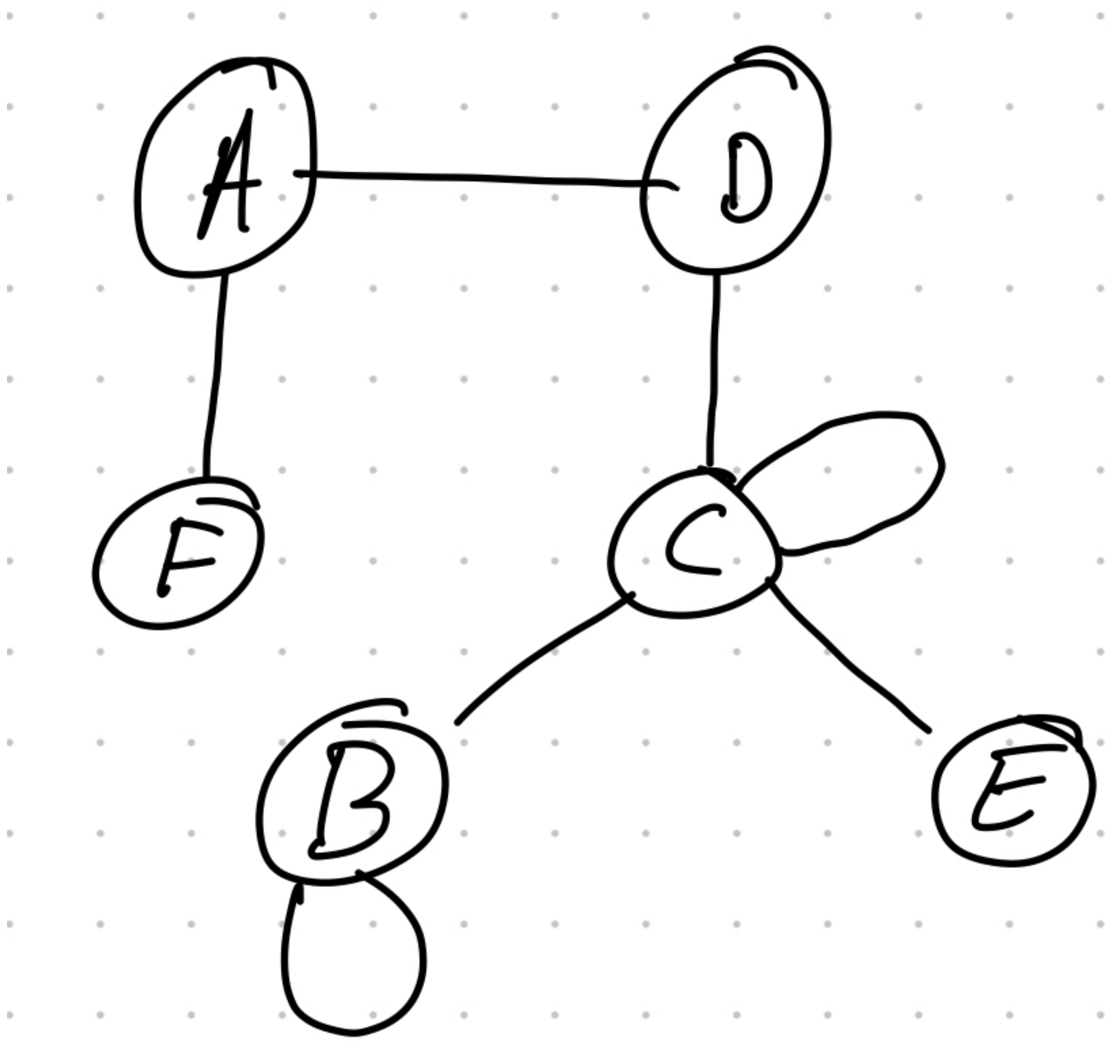
```
graph1 = { "a" : ["d","f"],
           "b" : ["c"],
           "c" : ["b", "c", "d", "e"],
           "d" : ["a", "c"],
           "e" : ["c"],
           "f" : ["a"]
         }
graph2 = { "a" : ["d","f"],
           "b" : ["c","b"],
           "c" : ["b", "c", "d", "e"],
           "d" : ["a", "c"],
           "e" : ["c"],
           "f" : ["a"]
         }
```

Draw a sketch of the above graphs and attach the screenshot here.

Graph1:



Graph2:



2. Develop a python program implementing Breadth First Search (BFS) algorithm. Perform a BFS on the above graphs from the previous question and attach the screenshot of the program and results here. Derive the time complexity of the BFS algorithm.

OUTPUT:

```
BFS on graph1:
a d f c b e
BFS on graph2:
a d f c b e
```

The time complexity of the BFS algorithm is O(V + E), where V is the number of vertices (nodes) and E is the number of edges in the graph. In the worst case, the BFS algorithm visits every vertex and edge once, which results in a linear time complexity.

3. Develop a python program implementing Depth First Search (DFS) algorithm. Perform a DFS on the above graphs from the previous question and attach the screenshot of the program and results

here. Derive the time complexity of the DFS algorithm.

OUTPUT:

```
DFS on graph1:
a d c b e f
DFS on graph2:
a d c b e f
```

The time complexity of the DFS algorithm is O(V + E), where V is the number of vertices (nodes) and E is the number of edges in the graph. In the worst case, the DFS algorithm visits every vertex and edge once, which results in a linear time complexity.

4. Discuss the advantages and disadvantages of BFS and DFS.

Advantages of BFS:

- **Shortest Path:** BFS is guaranteed to find the shortest path from the starting node to any other node in an unweighted graph. This makes it suitable for path-finding problems.
- **Completeness:** BFS is guaranteed to find a solution if one exists. It explores all nodes at the current level before moving to the next level, ensuring that all possibilities are considered.
- **Topological Sorting:** BFS can be used to perform topological sorting of a directed acyclic graph (DAG), which is essential in various applications, such as scheduling tasks or compiling code.
- **Memory Efficiency:** In some cases, BFS may be more memory-efficient than DFS, as it only needs to store the nodes at the current level.

Disadvantages of BFS:

- **Memory Consumption:** BFS can be memory-intensive, especially when dealing with large or densely connected graphs, as it needs to store all nodes at the current level.
- **Not Suitable for Very Deep Graphs:** In graphs with many levels or a deep structure, BFS might be impractical due to the large amount of memory required.

Advantages of DFS:

- **Memory Efficiency:** DFS is often more memory-efficient than BFS, especially for deep graphs or graphs with many branches, as it explores one branch fully before moving to the next.
- **Simplicity:** DFS is relatively simple to implement using recursion or a stack. It doesn't require as much overhead as BFS.
- **Topological Sorting:** Like BFS, DFS can be used to perform topological sorting of a DAG.
- **Detecting Cycles:** DFS is suitable for cycle detection in a graph. If a back edge is encountered while traversing, it indicates the presence of a cycle.

Disadvantages of DFS:

- **Completeness:** DFS may not find a solution even if one exists, especially in infinite graphs or graphs with infinite branches. It can get stuck in deep branches before exploring shallow ones.
- **Path Length:** DFS does not guarantee finding the shortest path, as it explores one branch deeply before exploring other branches.
- **Not Suitable for Unweighted Shortest Paths:** For unweighted graphs, DFS can be less efficient than BFS in finding the shortest path.

5. Discus in detail an application of BFS and DFS

BFS:

- In peer-to-peer network like bit-torrent, BFS is used to find all neighbor nodes

- Search engine crawlers are used BFS to build index. Starting from source page, it finds all links in it to get new pages
- Using GPS navigation system BFS is used to find neighboring places.
- In networking, when we want to broadcast some packets, we use the BFS algorithm.
- Path finding algorithm is based on BFS or DFS.
- BFS is used in Ford-Fulkerson algorithm to find maximum flow in a network.

DFS:

- If we perform DFS on unweighted graph, then it will create minimum spanning tree for all pair shortest path tree
- We can detect cycles in a graph using DFS. If we get one back-edge during BFS, then there must be one cycle.
- Using DFS we can find path between two given vertices u and v.
- We can perform topological sorting is used to scheduling jobs from given dependencies among jobs. Topological sorting can be done using DFS algorithm.
- Using DFS, we can find strongly connected components of a graph. If there is a path from each vertex to every other vertex, that is strongly connected.

Deliverables: Report, codes and the demonstration video (~3 minutes) For video demonstration, answer the following questions:

1. Explain the program for BFS and DFS including time complexity?

CODE

BFS.py

```
from collections import deque

def bfs(graph, start):
    # Create an empty set to keep track of visited nodes
    visited = set()
    # Create a queue for BFS using a deque
    queue = deque([start])
    # Mark the start node as visited
    visited.add(start)

    while queue:
        # Dequeue a node from the front of the queue
        node = queue.popleft()
        # Print the current node (you can modify this to store or process the node
as needed)
        print(node, end=" ")

        # Explore neighbors of the current node
        for neighbor in graph[node]:
            if neighbor not in visited:
                # Mark the neighbor as visited and enqueue it
                visited.add(neighbor)
                queue.append(neighbor)
```

```python
# Define graph1
graph1 = {
    "a": ["d", "f"],
    "b": ["c"],
    "c": ["b", "c", "d", "e"],
    "d": ["a", "c"],
    "e": ["c"],
    "f": ["a"]
}

# Perform BFS on graph1 starting from node 'a'
print("BFS on graph1:")
bfs(graph1, 'a')

# Define graph2
graph2 = {
    "a": ["d", "f"],
    "b": ["c", "b"],
    "c": ["b", "c", "d", "e"],
    "d": ["a", "c"],
    "e": ["c"],
    "f": ["a"]
}

# Perform BFS on graph2 starting from node 'a'
print("\nBFS on graph2:")
bfs(graph2, 'a')
```

DFS.py

```python
def dfs(graph, start, visited=None):
    # If 'visited' is not provided, initialize it as an empty set
    if visited is None:
        visited = set()

    # Add the current node to the 'visited' set and print it
    visited.add(start)
    print(start, end=" ")

    # Explore neighbors of the current node
    for neighbor in graph[start]:
        if neighbor not in visited:
            # Recursively call DFS for unvisited neighbors
            dfs(graph, neighbor, visited)

# Define graph1
graph1 = {
    "a": ["d", "f"],
    "b": ["c"],
    "c": ["b", "c", "d", "e"],
    "d": ["a", "c"],
    "e": ["c"],
    "f": ["a"]
}

# Perform DFS on graph1 starting from node 'a'
print("DFS on graph1:")
dfs(graph1, 'a')

# Define graph2
graph2 = {
    "a": ["d", "f"],
    "b": ["c", "b"],
    "c": ["b", "c", "d", "e"],
    "d": ["a", "c"],
    "e": ["c"],
    "f": ["a"]
}

# Perform DFS on graph2 starting from node 'a'
print("\nDFS on graph2:")
dfs(graph2, 'a')
```