Experiment No.2
Implement packet routing in a computer network using DFS and BFS.
Date of Performance: 29/01/2025
Date of Submission: 17/02/2025

Aim: Study and Implementation of Depth first search for problem solving.

Objective: To study the uninformed searching techniques and its implementation for problem solving.

Theory:

Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

- A search problem consists of:

- **A State Space.** Set of all possible states where you can be.
- **A Start State.** The state from where the search begins.
- **A Goal Test.** A function that looks at the current state returns whether or not it is the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state.
- This plan is achieved through search algorithms.

Depth First Search: DFS is an uninformed search method. It is also called blind search. Uninformed search strategies use only the information available in the problem definition. A search strategy is defined by picking the order of node expansion. Depth First Search (DFS) searches deeper into the problem space. It is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

The basic idea is as follows:

1. Pick a starting node and push all its adjacent nodes into a stack.
2. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
3. Repeat this process until the stack is empty.

However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Algorithm:

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

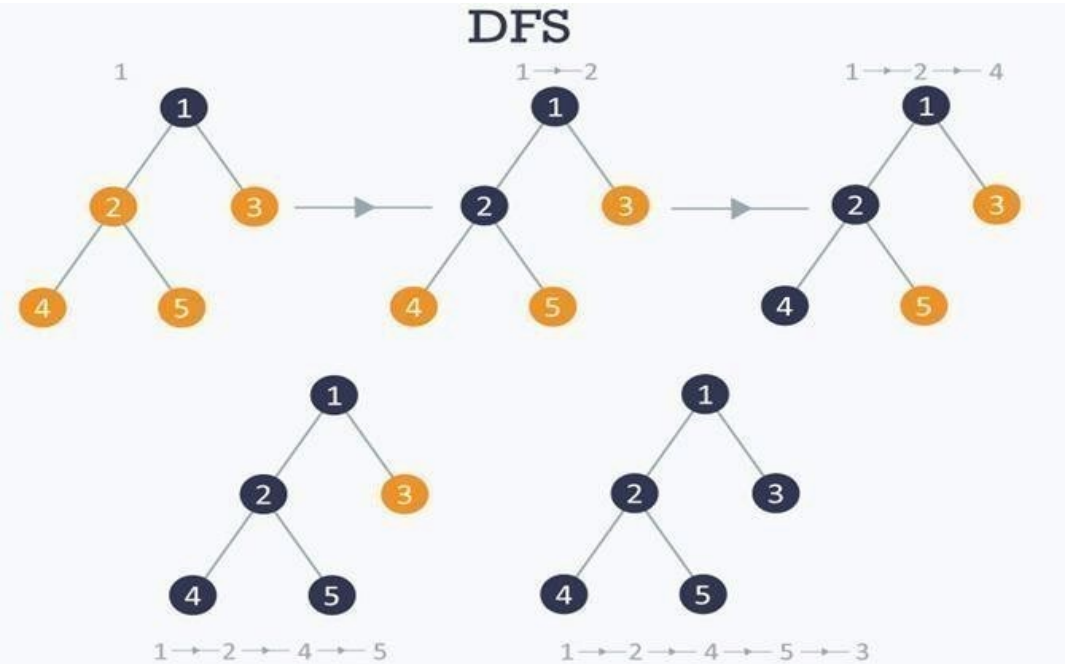
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles. The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Pseudocode:

```
DFS-iterative (G, s):
//Where G is graph and s is source vertex
let S be stack
      S.push( s )   //Inserting s in stack   mark
s as visited.  while (S is not empty):
//Pop a vertex from stack to visit next
v = S.top()
         S.pop()
//Push all the neighbours of v in stack that are not
visited   for all neighbours w of v in Graph G: if w
is not visited:
          S.push( w )
mark w as visited

DFS-recursive (G, s):
        mark s as visited          for all
neighbours w of s in Graph G:              if
w is not visited:                  DFS-
recursive(G, w)
```

DFS Working: Example



Path: 1 → 2→ 4→ 5→ 3

Searching Strategies are evaluated along the following dimensions:

1. **Completeness:** does it always find a solution if one exists?
2. **Time complexity:** number of nodes generated
3. **Space complexity:** maximum number of nodes in memory 4. **Optimality:** does it always find a least-cost solution?

Properties of depth-first search:

1. Complete: - No: fails in infinite-depth spaces, spaces with loops.
2. Time Complexity: O(bm)
3. Space Complexity: O(bm), i.e., linear space!
4. Optimal: No

Advantages of Depth-First Search:

1. Memory requirement is only linear with respect to the search graph.
2. The time complexity of a depth-first Search to depth d is O(b^d)
3. If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.

Disadvantages of Depth-First Search:

1. There is a possibility that it may go down the left-most path forever. Even a finite graph can generate an infinite tree.
2. Depth-First Search is not guaranteed to find the solution.
3. No guarantee to find a optimum solution, if more than one solution exists.

Applications:

How to find connected components using DFS?

A graph is said to be disconnected if it is not connected, i.e. if two nodes exist in the graph such that there is no edge in between those nodes. In an undirected graph, a connected component is a set of vertices in a graph that are linked to each other by paths.

Consider the example given in the diagram. Graph G is a disconnected graph and has the following 3 connected components.

- First connected component is 1 → 2 → 3 as they are linked to each other
- Second connected component 4 → 5
- Third connected component is vertex 6

Breadth First Search: BFS is a uninformed search method. It is also called blind search. Uninformed search strategies use only the information available in the problem definition. A search strategy is defined by picking the order of node expansion. It expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. It is very easily implemented by maintaining a queue of nodes. Initially the queue contains just the root. In each iteration, node at the head of the queue is removed and then expanded. The generated child nodes are then added to the tail of the queue.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

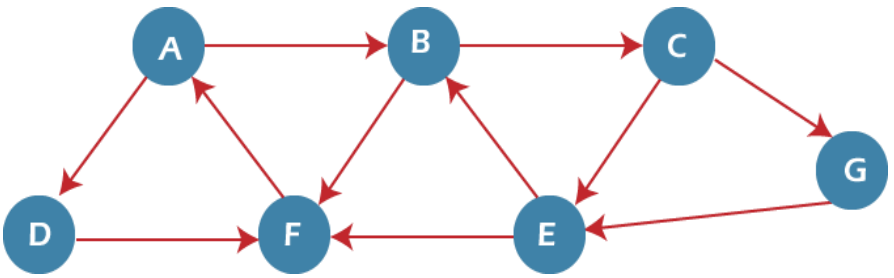
As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Question 1:

Apply DFS algorithm on given graph to find path from node A to node G.

Show and explain the status of all the nodes that are to be processed in stack STK and status of all the nodes that are already processed.



Solution:

1. Initialize
   - ➔ Start from A and push it onto the stack.
   - ➔ Stack (STK): [A]
   - ➔ Processed: []

2. Process A
   - ➔ Pop A, push its adjacent nodes (B, D, F) onto the stack.
   - ➔ Stack (STK): [B, D, F]
   - ➔ Processed: [A]

3. Process F
   - ➔ Pop F, push its adjacent node (E) onto the stack.
   - ➔ Stack (STK): [B, D, E]
   - ➔ Processed: [A, F]

4. Process E
   - ➔ Pop E, push its adjacent node (G) onto the stack.
   - ➔ Stack (STK): [B, D, G]

➔ Processed: [A, F, E]

5. Process G (Goal Reached)
   ➔ Pop G, and since it's the goal, stop the search.
   ➔ Stack (STK): [B, D] (Not relevant as we found the path)
   ➔ Processed: [A, F, E, G]

Step	Stack (STK)	Processed Nodes	Current Node	Action
1	[A]	[]	-	Start from A
2	[B, D, F]	[A]	A	Push B, D, F (A's neighbors)
3	[B, D]	[A, F]	F	Pop F, push E (F's neighbor)
4	[B, D, E]	[A, F]	E	Pop E, push G (E's neighbor)
5	[B, D, G]	[A, F, E]	G	Goal Reached!

Final Result

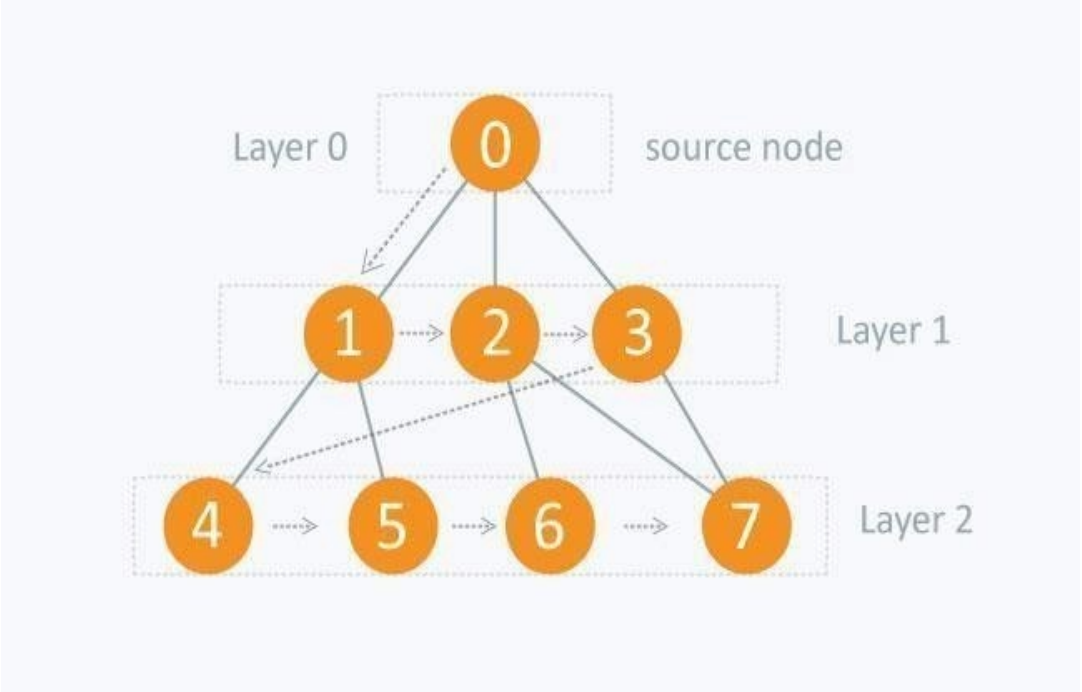
➔ Path Found: A → F → E → G
➔ Processed Nodes in Order: [A, F, E, G]
➔ Unprocessed Nodes (Remaining in Stack): [B, D] (Ignored as goal is found)

BFS Algorithm:

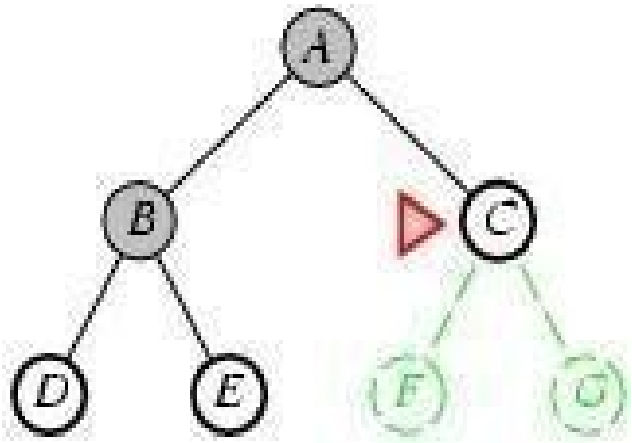
Pseudocode:

```
BFS (G, s)       //Where G is the graph and s is the source node let Q
be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour
vertices are marked.
       mark s as
visited.
      while ( Q is not empty)

          //Removing that vertex from queue,whose neighbour will be
visited now v = Q.dequeue(
           )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
          if w is not visited
                        Q.enqueue( w )              //Stores w in Q to
further visit its neighbour                          mark w as
visited.
```

Working of BFS:



Example: Initial Node: A Goal Node: C



Searching Strategies are evaluated along the following dimensions:

1. Completeness: does it always find a solution if one exists?
2. Time complexity: number of nodes generated
3. Space complexity: maximum number of nodes in memory 4. Optimality: does it always find a least-cost solution?

Properties of Breadth-first search:

1. **Complete**: - Yes: if b is finite.
2. **Time Complexity**: O(b^d+1)
3. **Space Complexity**: O(b^d+1)
4. **Optimal**: Yes

Advantages of Breadth-First Search:

1. Breadth first search will never get trapped exploring the useless path forever.
2. If there is a solution, BFS will definitely find it out.

3. If there is more than one solution then BFS can find the minimal one that requires less number of steps.

Disadvantages of Breadth-First Search:

1. The main drawback of Breadth first search is its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of BFS is O(bd).
2. If the solution is farther away from the root, breath first search will consume lot of time.

Applications:

How to determine the level of each node in the given tree?

As you know in BFS, you traverse level wise. You can also use BFS to determine the level of each node.

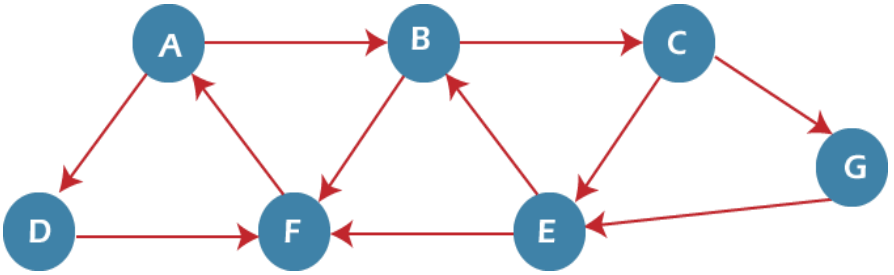
Question 2:

Apply BFS algorithm on given graph to find path from node A to node E.

Show and explain the status of both the queues Q1 and Q2.

Q1 holds all the nodes that are to be processed

Q2 holds all the nodes that are processed and deleted from Q1.



Solution:

Step	Q1 (To Be Processed)	Q2 (Processed)	Current Node	Action
1	[A]	[]	-	Start from A
2	[D, F, B]	[A]	A	Push D, F, B (A's neighbors)
3	[F, B]	[A, D]	D	Pop D (No new neighbors)
4	[B, E]	[A, D, F]	F	Pop F, push E (F's neighbor)
5	[E]	[A, D, F, B]	B	Pop B, push E (already in Q1)
6	[]	[A, D, F, B, E]	E	Goal Reached!

Code:

```
import matplotlib.pyplot as plt import
networkx as nx
from collections import deque


# Create the network graph network_graph
= nx.Graph()
```

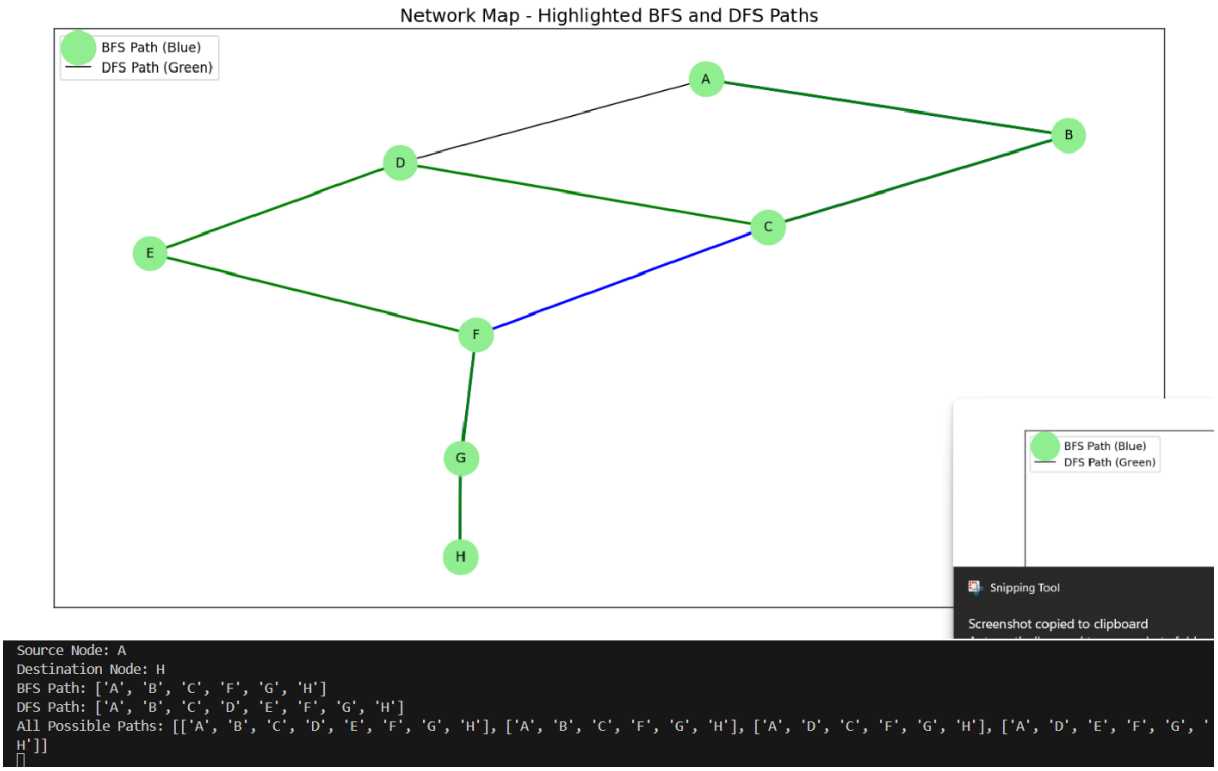
```
# Define nodes using alphabets
nodes = ["A", "B", "C", "D", "E", "F", "G", "H"]

network_graph.add_nodes_from(nodes)

# Define edges between the nodes edges
= [
    ("A", "B"),
    ("B", "C"),
    ("C", "D"),
    ("D", "E"),
    ("E", "F"),
    ("F", "G"),
    ("G", "H"),
    ("A", "D"), # Direct connection between A and D
    ("F", "C") # Direct connection between F and C
]

network_graph.add_edges_from(edges)

# Visualize the network graph plt.figure(figsize=(8, 6)) pos =
nx.spring_layout(network_graph, seed=42) nx.draw_networkx_nodes(network_graph,
pos, node_size=700, node_color="lightgreen") nx.draw_networkx_edges(network_graph,
pos, style='solid', edge_color="black") nx.draw_networkx_labels(network_graph, pos,
font_size=10, font_color="black")
```

```
# Highlight the best path (either BFS or DFS) def
highlight_best_path(path, color="red"):
    edge_list = list(zip(path, path[1:])) # Create edges from consecutive nodes in path
    nx.draw_networkx_edges(network_graph, pos, edgelist=edge_list, edge_color=color,
width=2)


# Define BFS algorithm to find the shortest path def
bfs(network_graph, start, goal):
    visited = set() queue =
    deque([[start]]) while
    queue: path =
    queue.popleft() node =
    path[-1]


        if node == goal:
            return path


        if node not in visited:
            visited.add(node)


            for neighbor in network_graph.neighbors(node):
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)


    return None
```

```
# Define DFS algorithm to find the first path def
dfs(network_graph, start, goal, path=None, visited=None):
    if path is None:
        path = [start] if
    visited is None:
       visited = set()


    visited.add(start)
    if start == goal:
        return path


    for neighbor in network_graph.neighbors(start):
        if neighbor not in visited:
            new_path = dfs(network_graph, neighbor, goal, path + [neighbor], visited)
            if new_path:
                return new_path


    return None


# Test with nodes "A" and "H"
source_node = "A" destination_node
= "H"


print(f"Source Node: {source_node}") print(f"Destination
Node: {destination_node}")
```

```
# BFS and DFS paths bfs_path = bfs(network_graph,
source_node, destination_node) print("BFS Path:", bfs_path)

dfs_path = dfs(network_graph, source_node, destination_node) print("DFS
Path:", dfs_path)

# All possible paths between the source and destination all_paths =
list(nx.all_simple_paths(network_graph, source_node, destination_node)) print("All
Possible Paths:", all_paths)


# Highlight the best path for BFS (blue) and DFS (green) highlight_best_path(bfs_path,
color="blue") # Highlight BFS path in blue highlight_best_path(dfs_path,
color="green") # Highlight DFS path in green

# Add a legend to explain the colors
plt.legend(["BFS Path (Blue)", "DFS Path (Green)"], loc="upper left")

# Display the graph with the highlighted best paths plt.title("Network
Map - Highlighted BFS and DFS Paths", fontsize=14) plt.show()
```

Output:



Conclusion:

In conclusion of computer network packet routing, both DFS and BFS have their strengths and weaknesses. DFS is effective for deep exploration but may not always find the shortest path, whereas BFS guarantees finding the shortest path but requires more memory and time in large networks.**DFS** is suitable when memory is a concern, and we don't necessarily require the shortest path. It's useful in more exploratory routing scenarios.**BFS** is ideal when the shortest path is crucial, and it ensures the most efficient route in terms of hops.

In real-world networks, however, more sophisticated algorithms like Dijkstra's or A* are typically used to find the most optimal routes in terms of both time and space, as they take into account weights and other factors.