22. To go about the iterative implementation, we can use a stack which replaces the stack containing the recursive calls

```
DFS(G)
// G is a graph (V, E)
// Each vertex u contains fields: color, d, f, π
   and all have their usual meaning
// output: complete discovery time, finish time and predecessor
   in DFS forest for each vertex
    for each vertex u in G.V
        u.color = WHITE
        u.π = NIL

    time = 0
    st = ϕ  // empty stack

    for each vertex u in G.V
        if u.color == WHITE
            u.color = GRAY
            u.d = time + 1
            push(st, u)

            while st != ϕ
                curr = top(st)
                nextUnvisited = findNextUnvisited(curr)

                if nextUnvisited != null
                    nextUnvisited.color = GRAY
                    nextUnvisited.π = curr
                    nextUnvisited.d = time + 1
                    push(st, nextUnvisited)
```

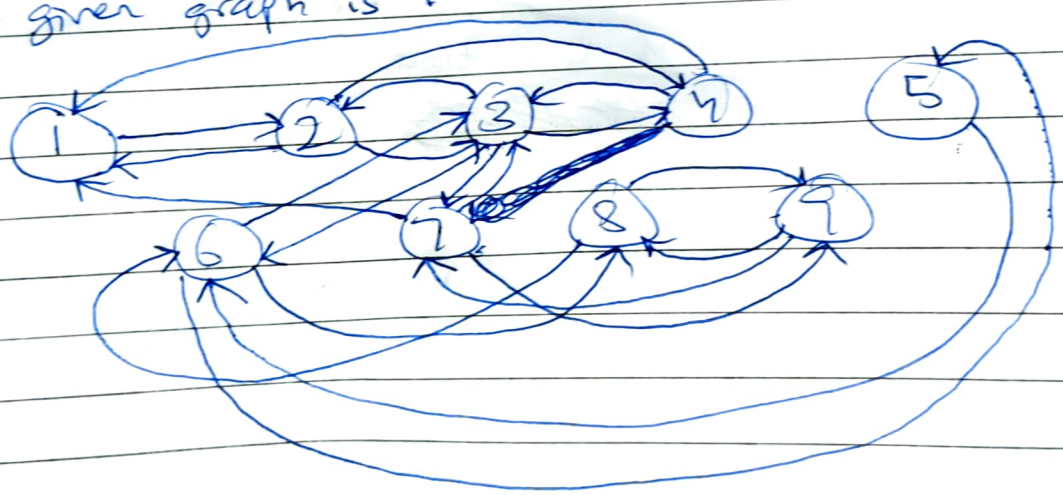
```
else
    curr.color = BLACK
    time = time + 1
    curr.f = time
    pop(st)

findNextUnvisited(u) // output : return unvisited neighbour
    for each vertex v in G.Adj[u]
        if v.color == WHITE
            return v

    return null
```

In the pseudocode, the line 'nextUnvisited.π = curr' assigns the current vertex as predecessor of the next unvisited neighbour and this allows us to keep track of the parent child relationship.

The given graph is :



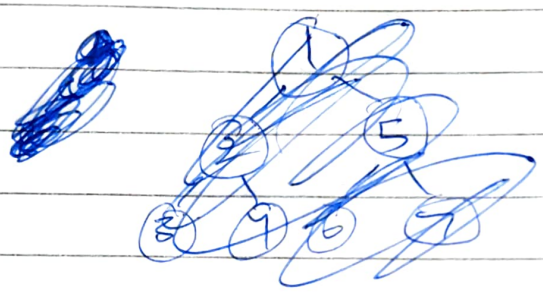

→ 1 is visited
→ 2 is visited
→ 3 is visited
→ 4 is visited
→ 4 is blackened
→ 7 is visited → 6 is blackened
→ 9 is visited → 8 is blackened
→ 8 is visited → 9 is blackened
→ 6 is visited → 7 is blackened
→ 5 is visited → 3 is blackened
→ 5 is blackened → 2 is blackened
→ 1 is blackened

then BFS

→ a large sparsely connected graph where target vertex is far away from source vertex

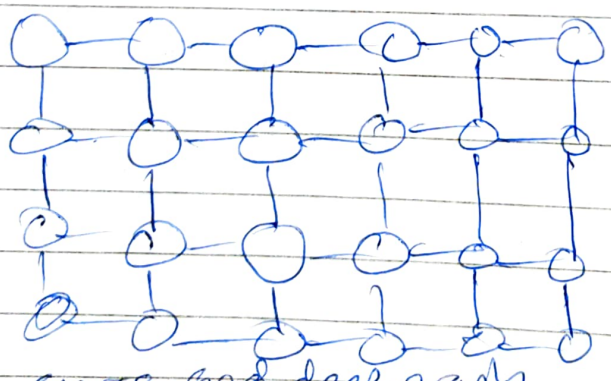
DFS is more efficient when there is a need to explore deep branches and traverse long paths in sparsely connected graphs. For BFS, it guarantees finding shortest path and is suitable when multiple connecting paths exist and focus is on shortest path.

eg.




if going from 1 to target 7, 1 → 2 → 3 → 4 → 5 → 6 → 7

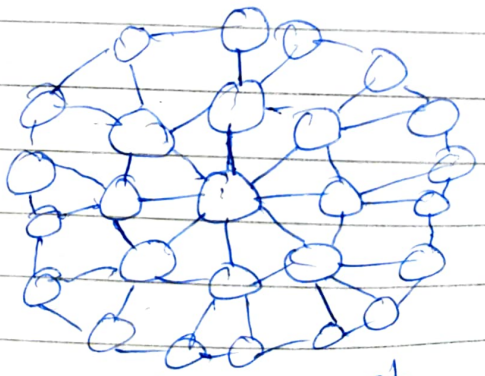
- DFS is better



sparse and deep graph

→ recursively traverse all levels would be faster than assigning levels and then finding shortest path.

- BFS is better



dense graph

→ assigning levels would be helpful, especially if source is towards the centre of the graph, hence BFS

24. DFS(G)

forwardEdges = ∅ // empty list

for each vertex u in G.V
    u.color = WHITE
    u.π = NIL

for each vertex u in G.V
    if u.color == WHITE
        DFS-VISIT-Forward-Edges(G, u, forwardEdges)

print forwardEdges

DFS-VISIT-Forward-Edges(G, u, forwardEdges)

u.color = GRAY

for each vertex v in G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT-Forward-Edges(G, v, forwardEdges)
    else if v.color == GRAY
        forwardEdges.append((u, v))

u.color = BLACK