Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

Subject: Artificial Intelligence (DJ19DSC502) AY: 2023-24

Experiment 2 (Uninformed Search)

D 12

60009210105 **Amitesh Sawarkar**

Aim: Implement Depth First Iterative Deepening to find the path for a given planning problem.

Theory:

Solving a problem by search is solving a problem by trial and error. Several real-life problems can be modelled as a state-space search problem.

1. Choose your problem and determine what constitutes a STATE (a symbolic representation of the state- of-existence).

2. Identify the START STATE and the GOAL STATE(S).

3. Identify the MOVES (single-step operations/actions/rules) that cause a STATE to change.

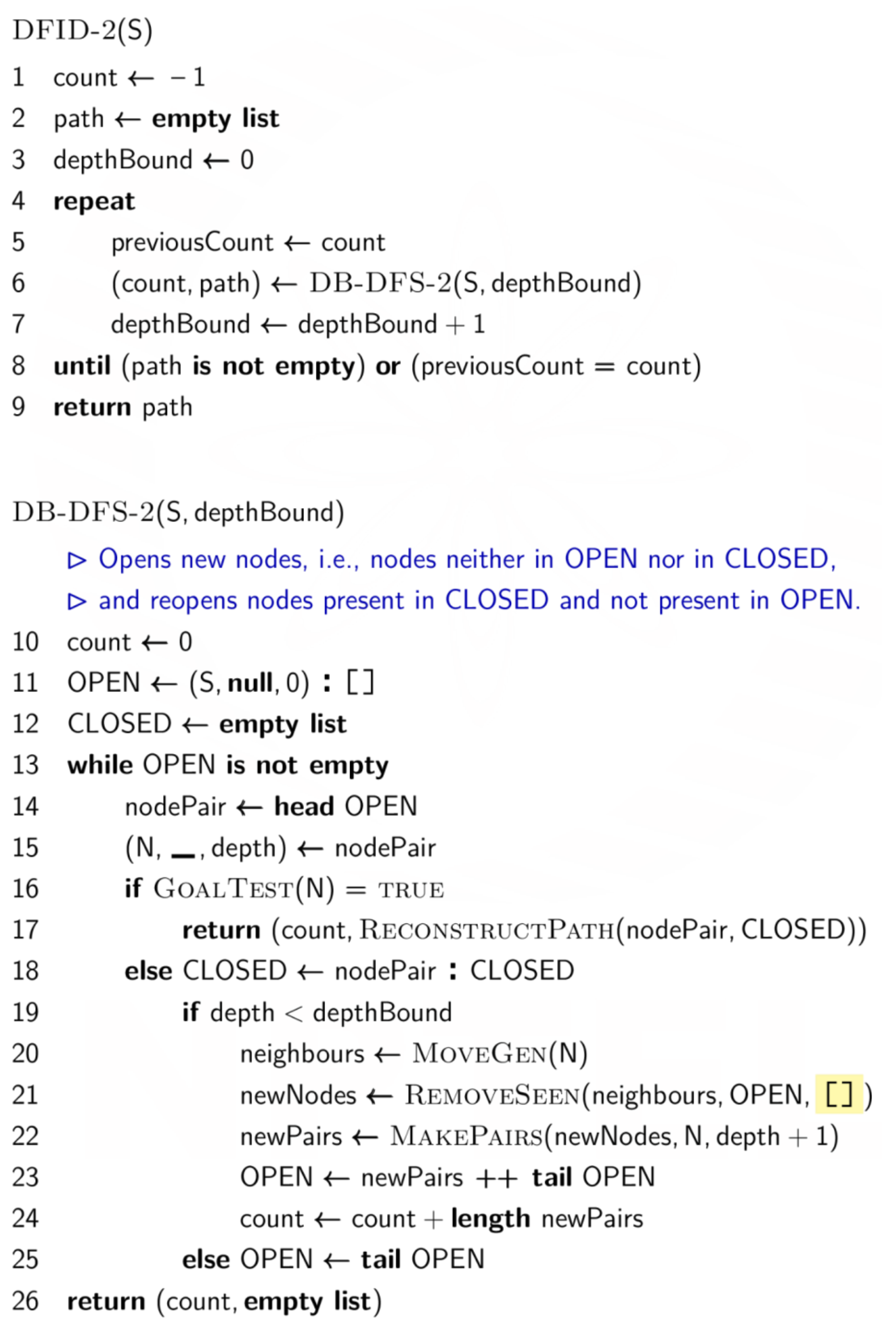
4. Write a function that takes a STATE and applies all possible MOVES to that STATE to produce a set of NEIGHBOURING STATES (exactly one move away from the input state). Such a function (state-transition function) is called MoveGen. MoveGen embodies all the single-step operations/actions/rules/moves possible in a given STATE. The output of MoveGen is a set of NEIGHBOURING STATES. MoveGen: STATE -
-> SET OF NEIGHBOURING STATES. From a graph theoretic perspective the state space is a graph, implicitly defined by a MoveGen function. Each state is a node in the graph, and each edge represents one move that leads to a neighbouring state. Generating the neighbours of a state and adding them as candidates for inspection is called "expanding a state". In state space search, a solution is found by exploring the state space with the help of a MoveGen function, i.e., expand the start state and expand every candidate until the goal state is found.

State spaces are used to represent two kinds of problems: configuration and planning problems.

1. In configuration problems the task is to find a goal state that satisfies some properties.

2. In planning problems the task is to find a path to a goal state. The sequence of moves in the path constitutes a plan.

Algorithm DFID



```
DFID-2(S)
count ← −1
path ← empty list
depthBound ← 0
repeat
    previousCount ← count
    (count, path) ← DB-DFS-2(S, depthBound)
    depthBound ← depthBound + 1
until (path is not empty) or (previousCount = count)
return path

DB-DFS-2(S, depthBound)
    ▷ Opens new nodes, i.e., nodes neither in OPEN nor in CLOSED,
    ▷ and reopens nodes present in CLOSED and not present in OPEN.
count ← 0
OPEN ← (S, null, 0) : []
CLOSED ← empty list
while OPEN is not empty
    nodePair ← head OPEN
    (N, _, depth) ← nodePair
    if GoalTest(N) = TRUE
        return (count, ReconstructPath(nodePair, CLOSED))
    else CLOSED ← nodePair : CLOSED
        if depth < depthBound
            neighbours ← MoveGen(N)
            newNodes ← RemoveSeen(neighbours, OPEN, [])
            newPairs ← MakePairs(newNodes, N, depth + 1)
            OPEN ← newPairs ++ tail OPEN
            count ← count + length newPairs
        else OPEN ← tail OPEN
return (count, empty list)
```

Auxiliary Functions for DFID

```
MakePairs(nodeList, parent, depth)
if nodeList is empty
    return empty list
else nodePair ← (head nodeList, parent, depth)
    return nodePair : MakePairs(tail nodeList, parent, depth)
```

```
ReconstructPath(nodePair, CLOSED)
SkipTo(parent, nodePairs, depth)
    if (parent, _, depth) = head nodePairs
        return nodePairs
    else return SkipTo(parent, tail nodePairs, depth)

(node, parent, depth) ← nodePair
path ← node : []
while parent is not null
    path ← parent : path
    CLOSED ← SkipTo(parent, CLOSED, depth – 1)
    (_, parent, depth) ← head CLOSED
return path
```

Lab Assignment to do:
Select any one problem from the following and implement DFID to find the path from start state to goal state. Analyse the Time and Space complexity. Comment on Optimality and completeness of the solution.

Problem 3: Graph

```python
from collections import defaultdict

def dfs(graph, node, goal, depth, visited):
    if depth == 0:
        return False
    if node == goal:
        return True

    mark_visited(visited, node)

    for neighbor in generate_moves(graph, node):
        if not visited[neighbor]:
            if dfs(graph, neighbor, goal, depth - 1, visited):
                return True

    return False
def dfid(graph, start, goal):
    max_depth = 0

    while True:
        visited = defaultdict(bool)
        if dfs(graph, start, goal, max_depth, visited):
            return reconstruct_path(graph, start, goal, visited)
        max_depth += 1

def generate_moves(graph, node):
    return graph[node]

def set_goal(goal_node):
    return goal_node
```

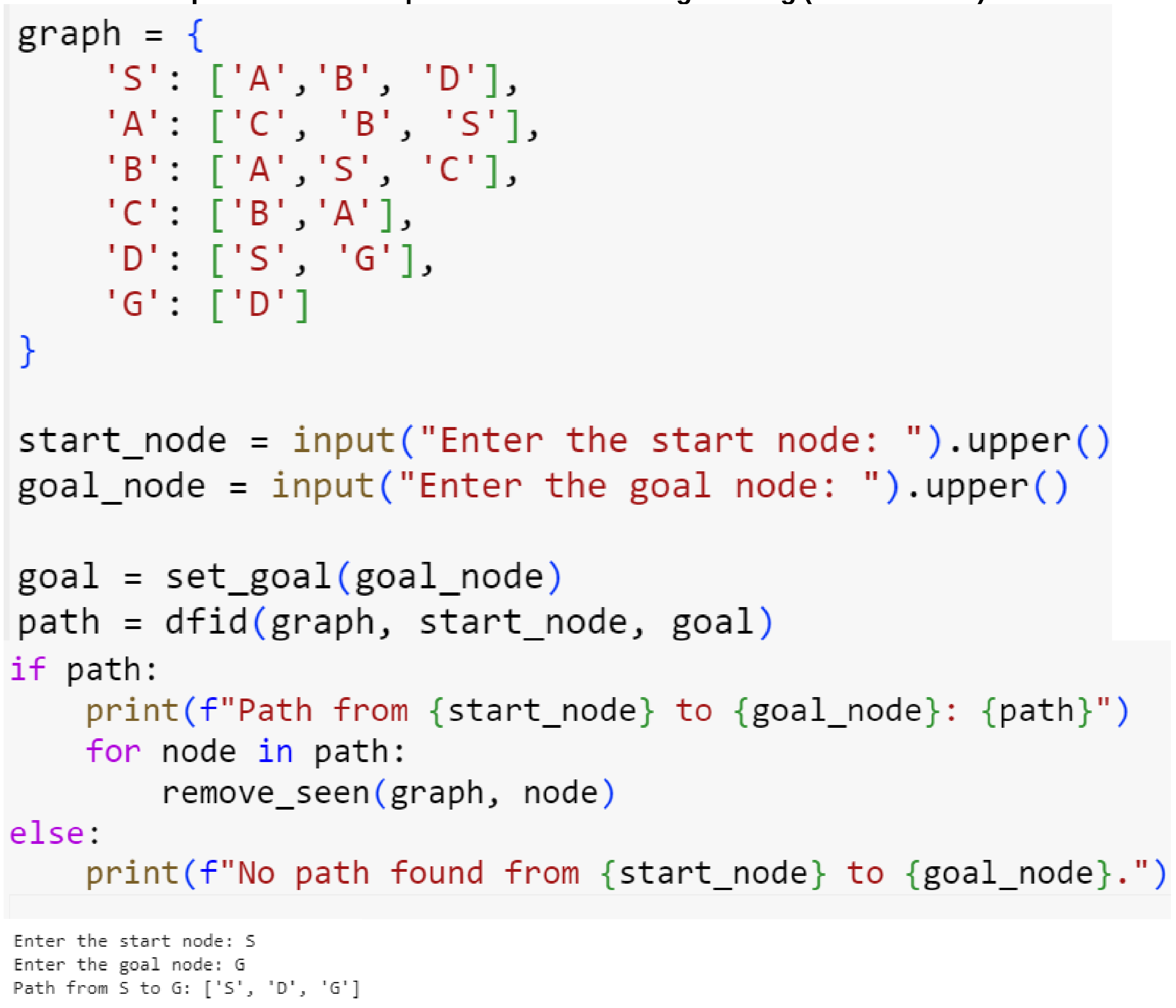
```python
def reconstruct_path(graph, start, goal, visited):
    path = []
    current = goal

    while current != start:
        path.append(current)
        for neighbor in graph[current]:
            if visited[neighbor]:
                current = neighbor
                break

    path.append(start)
    path.reverse()
    return path

def mark_visited(visited, node):
    visited[node] = True

def remove_seen(visited, node):
    visited[node] = False
```



```
graph = {
    'S': ['A','B', 'D'],
    'A': ['C', 'B', 'S'],
    'B': ['A','S', 'C'],
    'C': ['B','A'],
    'D': ['S', 'G'],
    'G': ['D']
}

start_node = input("Enter the start node: ").upper()
goal_node = input("Enter the goal node: ").upper()

goal = set_goal(goal_node)
path = dfid(graph, start_node, goal)
if path:
    print(f"Path from {start_node} to {goal_node}: {path}")
    for node in path:
        remove_seen(graph, node)
else:
    print(f"No path found from {start_node} to {goal_node}.")
```

```
Enter the start node: S
Enter the goal node: G
Path from S to G: ['S', 'D', 'G']
```