Lab= 2

"8-Puzzle Problem"

Date 25/10/24
Page

```
from collections import deque

def print_puzzle(puzzle):
    for row in puzzle:
        print(' '.join(str(x) if x!=0 else '_' for x
        in row))
    print() # gives newline


def find_blank(puzzle):
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] ==0:
                return i,j


def get_neighbours(puzzle):
    neighbours = [] # empty list
    blank_x, blank_y = find_blank(puzzle)
    moves = [(0,1),(0,-1),(1,0),(-1,0)]
        # right, left, down, up

for dx, dy in moves:
    new_x = blank_x + dx
    new_y = blank_y + dy

    if 0<=new_x<3 and 0<=new_y<3:

    new_puzzle = [row[:] for row in puzzle]

    new_puzzle[blank_x][blank_y], new_puzzle[new_x]
    [new_y] = new_puzzle[new_x][new_y],
                new_puzzle[blank_x][blank_y]
    neighbours.append(new_puzzle)
    return neighbours

def bfs(start, goal):
    queue = deque([start]) # double ended queue for BFS
    visited = set()
    visited.add(tuple(map(tuple, start))) # Add initial state
    to map
    # To keep track the path
    parent_map = {tuple(map(tuple, start)): None}
    while queue:
        current = queue.popleft()
        if current == goal:
            return current, parent_map
    # Explore neighbours
    for neighbor in get_neighbours(current):
        neighbor_tuple = tuple(map(tuple, neighbor))
        if neighbor_tuple not in visited:
            visited.add(neighbor_tuple)
            queue.append(neighbor)
            parent_map[neighbor_tuple] = tuple(map
                (tuple, current))

    return None, parent_map


def get_solution_path(start, goal, parent_map):
    path = []
    current = tuple(map(tuple, goal))
    while current is not None:
        path.append(current)
        current = parent_map[current]

    return path[::-1] # reverse path to get
    it from start to goal


def solve_8_puzzle():
    print("Enter starting state of 8-puzzle (use 0
    for blank space):")
```

```
start_state = []
for i in range(3):
    row = input(f"Enter row{i+1} (3 nos separated by space):")
    start_state.append([int(x) for x in row.split()])

goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

print("Starting puzzle:")
print_puzzle(start_state)

solution, parent_map = bfs(start_state, goal_state)

if solution:
    print("Solution found:")
    print_puzzle(solution)

    show_transition = input("Do you want to see the transition states from start to goal? (Yes/No):").strip().lower()
    if show_transition == 'yes':
        path = get_solution_path(start_state, solution, parent_map)

        print("Transition states:")
        for state in path:
            print_puzzle(state)
    else:
        print("No solution found")
```

solve_8_puzzle()

Algorithm

```
function BFS(start, goal)
    queue = empty deque()
    visited = empty set()
parent_map = empty dictionary()

    queue.append(start)
    visited.add(start)
    parent_map[start] = None

while queue is not empty:
    current = queue.popleft()
if current is goal
    return current, parent_map

for each neighbour in get_neighbour(current):
    if neighbour is not in visited:
        visited.add(neighbour)
        queue.append(neighbour)
        parent_map[neighbour] = current

return none, parent_map

function get_neighbour(puzzle):
    neighbours = empty list
(blank_x, blank_y) = find_blank(puzzle)
moves = [(0,1),(0,-1),(1,0),(-1,0)]

for (dx, dy) in moves:
    new_x = blankx + dx
    new_y = blanky + dy

if (new_x, new_y) is within puzzle:
    new_puzzle = copy(puzzle)
```

Swap new_puzzle[blank_x][blank_y] with new_puzzle[new_x][new_y]

Date / /
Page

neighbours.append(new_puzzle)

return neighbours:

```
function find_blank(puzzle):
    for row in range(0,3)
        for col in range(0,3)
            if puzzle[row][col] == 0
                return(row, col)

function getsolution_path(start, goal, parent_map):
    path = empty list
    current = goal

    while current is None:
        path.append(current)
        current = parent_map[current]

    return reversed(path)

function solvedpuzzle()
    print("Take Input")

start_state = empty list
for i in range(0,3):
    row = input
    start_state.append([int(x) for x in row.])

goal_state = [[1,2,3],[4,5,6],[7,8,9]]

print_puzzle(start_state)
```

Date / /
Page

```
solution, parent_map = BFS(start state, goal_state)
    if solution is not None:
        print_puzzle(solution)

function print_puzzle(puzzle):
    for row in puzzle:
        print()
```

Output

Enter starting state (use 0 for blank space)
Enter row1: 1 2 3
Enter row2: 5 4 6
Enter row3: 0 8 7

Starting puzzle	solution found
1 2 3	1 2 3
5 4 6	4 5 6
_ 8 7	7 8 _

Iterative deepening Search algorithm

Date / /
Page

```
class Graph:
    function __init__():
        graph = empty dictionary

    function add_edge (u,v):
        if u is not in graph:
            graph[u] = empty list
        graph[u].append(v)

function depthsearch (source, target, limit):
    if source == target:
        return True
    if limit <= 0
        return False
    if source in graph:
        for each neighbour in graph[source]:
            if depthSearch (neighbour, target, limit-1)
            return True

    return False

function iterativedeep(src, target, max-depth):
    for depth in range(0, max-depth + 1):
        print ("depth limit")
        if depthSearch (src, target, depth)
            print "Target" + target + " found at
                depth level" + depth

            return True
    return False

function main():
    graph = new.graph()
    max-edge = input
```

Date / /
Page

```
print to enter edge pair (Source, destn)
for i in range (0, max-edges):
    u, v = input (" Enter edge: "). split ()
    graph.add-edge (int(u), int(v));

src = input ("enter source node:")
target = input (" enter target node")
max_depth = Input ["enter max depth limit")

if iterativedeepning (int(src), int(target), int(max_depth));
    print "Reachable target"
else
    print "not reachable target"
```

Output

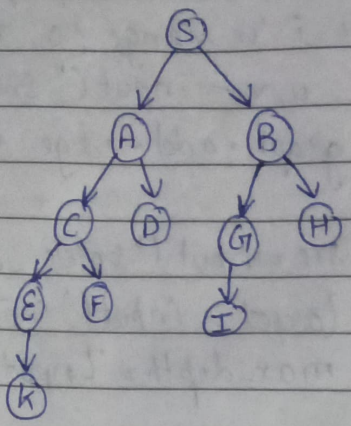
```
Enter no. of edges in graph: 5
Enter each edge as a pair (source, dest):
0 1
0 2
1 3
1 4
2 5
Enter source node: 0
Enter target node: 5
Enter max depth limit: 3
Checking with depth limit: 0
Checking with depth limit: 1
Checking with depth limit: 2
Target 5 found at depth level: 2

Target 5 is reachable from source 0 with depth 3
```

Q) Search Problem

Date __/__/__
Page

(1) Perform BFS

1. Visit S



frontier | S |

explored | |

(2) F → | A | B |

E → | S |

(3) F → | B | C | D |

E → | S | A |

(4) F → | C | D | G | H |

E → | S | A | B |

(5) F → | D | G | H | E | F |

E → | S | A | B | C |

(6) F → | G | H | E | F |

E → | S | A | B | C | D |

(7) F → | H | E | F | I |

E → | S | A | B | C | D | G |

(8) F → | E | F | I |

E → | S | A | B | C | D | G | H |

(9) F → | F | I | K |

E → | S | A | B | C | D | G | H | E |

(10) F → |

E → | S | A | B | C | D | G | H | E | I | K |

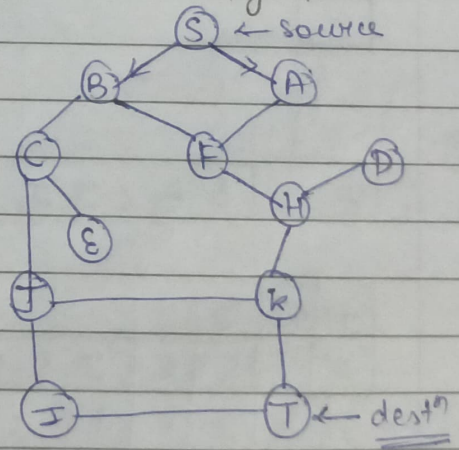
Dest^{n} (K) found

Date __/__/__
Page _____

Pseudo code

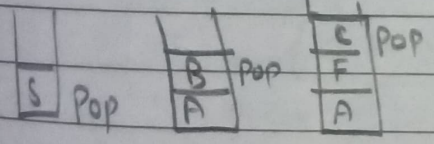
```
BFS ( Graph, startNode)
(1) queue = empty queue          // initialization
    visited = empty set

(2) queue.enqueue (start Node)
    visited.add ( startNode)

(3) while queue is not empty:
        currentNode = queue.dequeue ()
        print (currentNode)

(4) for each neighbour in neighbour of currentNode:
        if neighbour is not in visited:
            queue.enque (neighbor)
            visited.add ( neighbor)
```

Q2 Perform DFS on both Tree search & graph search

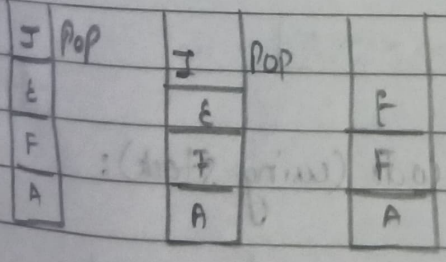


1) Tree search

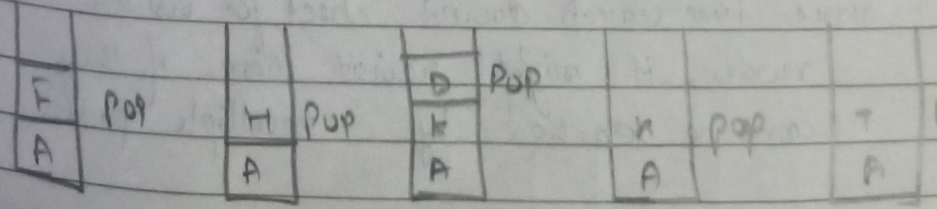
2)



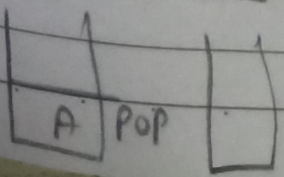
3)



4)



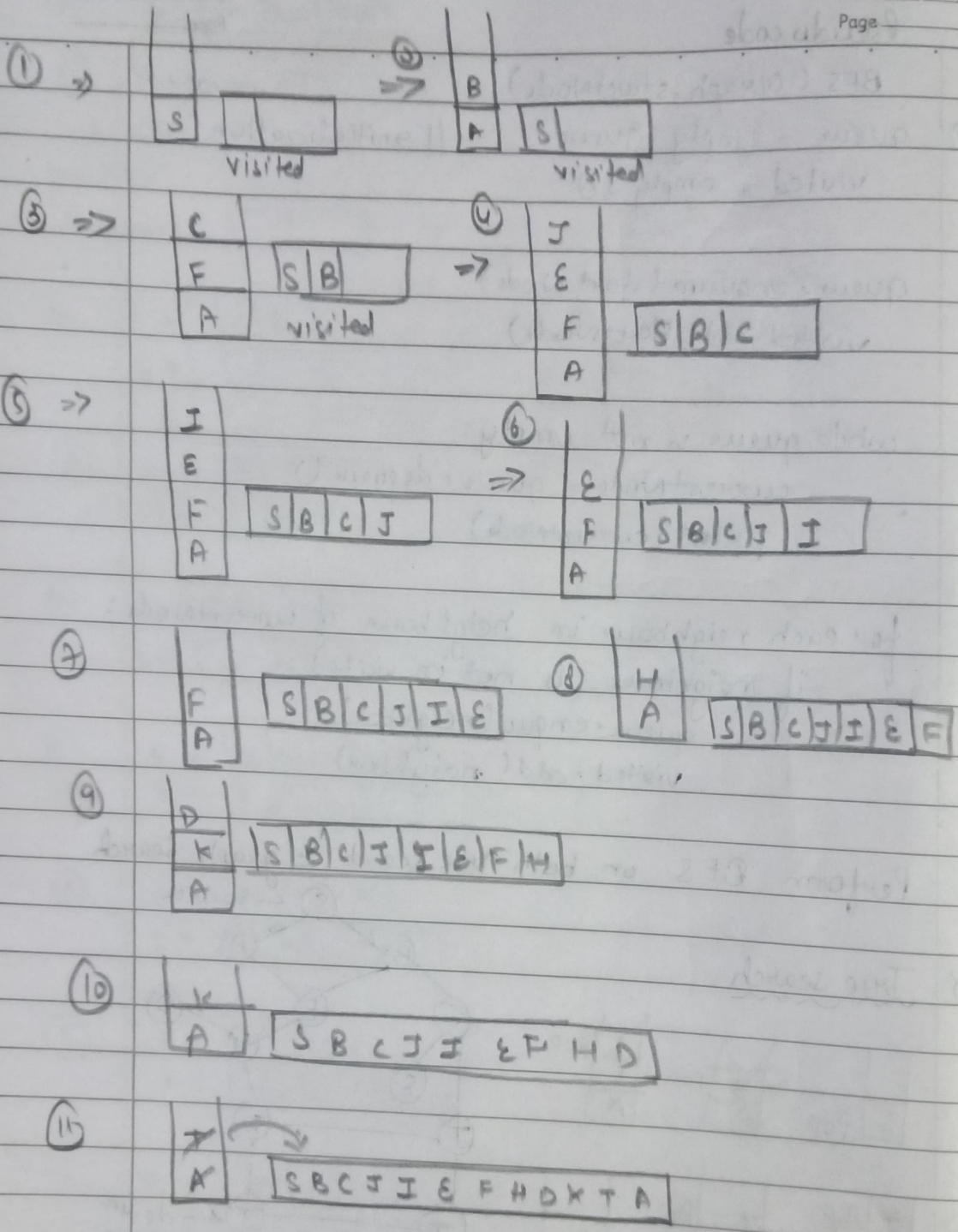
5)



final search →
S → B; C, I, J, E, F, H, D, H, K, A, T

(2) Graph search

Date __/__/__
Page



DFS Tree search Pseudocode (using stack):

Since tree search doesn't check for visited nodes, it might revisit them if they can be reached by multiple paths

```
DFS_TreeSearch(Graph, startNode):
    stack ← empty stack
    stack.push(startNode)

    while stack is not empty:
        node ← stack.pop()
        print(node)
        for each neighbor of node:
            stack.push(neighbor)
```

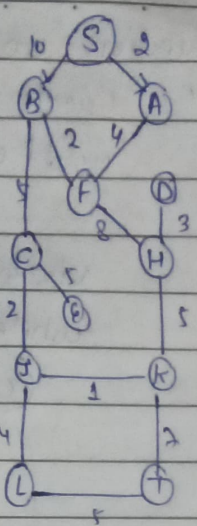
DFS Graph Search Pseudocode (using stack):

Graph search checks for visited nodes to avoid revisiting the same node, preventing cycles or multiple visits.

```
DFS_GraphSearch(Graph, startNode):
    stack ← empty stack
    visited ← empty set

    stack.push(startNode)
    visited.add(startNode)

    while stack is not empty:
        node ← stack.pop()
        print(node)

        for each neighbor of node:
            if neighbor is not in visited:
                stack.push(neighbor)
                visited.add(neighbor)
```

3.) perform Uniform Cost Search

Date __/__/__
Page _____



(1) Frontier → | S |
Cost | 0 |
Explored | |

(2) F → | A | B |
C → | 2 | 10 |
E → | S |

(3) F → | B | F |
C → | 10 | 6 |
E → | S | A |

(4) F → | B | H |
C → | 10 | 14 |
E → | S A F |

(5) F → | H | C |
C → | 14 | 15 |
E → | S A F B |

(6) F → | C | D | K |
C → | 15 | 17 | 19 |
E → | S A F B H |

(7) F → | D | K | E | J |
C → | 17 | 19 | 20 | 27 |
E → | S A F B H C |

(8) F → | K | E | J |
C → | 19 | 20 | 22 |
E → | S A F B H C D |

(9) F → | E | J | T |
C → | 20 | 27 | 26 |
E → | S A F B H C D K |

(10) F → | J | T |
C → | 27 | 26 |
E → | S A F B H C D K E |

(11) F → | L | T |
C → | 31 | 26 |
E → | S A F B H C D K E J |

(12) F | L
C | 31
E → | S A F B H C D K E J T |

⇒ E = | S A F B H C D K E J T L |

(T) reached

min_path ⇒ S A F H K T → Cost = 2+4+8+5+7

⇒ 26

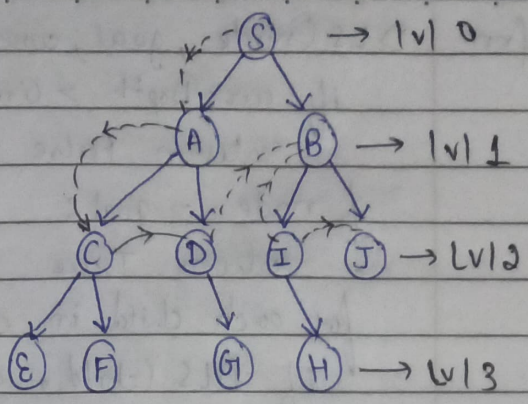
Date __/__/__
Page

Pseudocode

```
function uniformCostSearch(start, goal):
    Create a priority queue called frontier
    add(0, start, None) to frontier

    visited = set()
    while frontier is not empty:
        (current_cost, current_state, prev_state) =
                frontier.extract_min()

        if current_state is goal:
            return reconstruct_path(prev_state)

        if current_state is not in visited:
            visited.add(current_state)

        for each neighbor n of current_state:
            if n not in visited:
                cost_n = current_cost + cost_to_neighbor
                                (current_state, n):
                frontier.add(cost_n, n, current_state)

    return failure
```

4. Depth Limited search

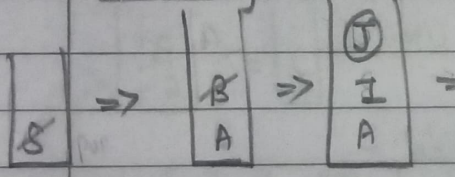
Date __/__/__
Page _____



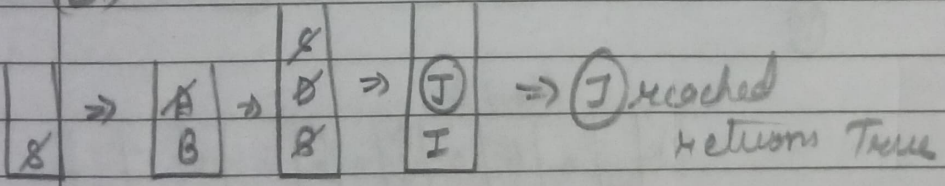
pseudocode

```
fxn DepthLS(node, depth, limit):
    if Depth > limit:
        return False
    if node == goal:
        return True
    for each child in children(node):
        if DepthLS(child, depth+1, limit):
            return True
    return False
```

Working: source → S Dest' → J DLimit = 2



(OR)



5) Iterative Deepening Search

Pseudocode

```
fxn IDS(root, goal):
    depth = 0
    while True:
        result = DLS(root, goal, 0, depth)
        if result == True:
            return True
        depth = depth + 1
```

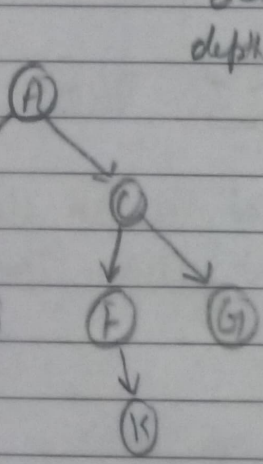
Date / /
Page

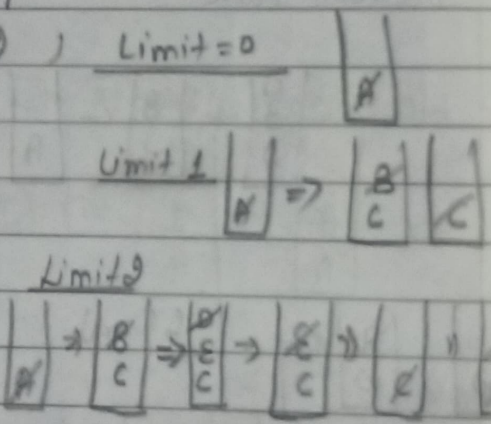
```
fxn DLS(node, goal, current_depth, limit):
    if currDepth > limit:
        return False
    if node == goal:
        return True
    for each child in children(node):
        if DLS(child, goal, curr_depth + 1, limit):
            return True
    return false
```

Working → Source → A
Destⁿ → G
depthLimit → 3

LIFO structure

Limit = 0





G reached return True
Search stop