NAME	KunalParmar
SEAT NO	31010921113
SUBJECT	AI JOURNAL

INDEX

Practical	DFS
1	BFS
Practical	Shortest path program
2	N-queen problem
Practical	Tower Hanoi problem
3	Jug water problem
Practical	Missionaries and Cannibals
4	Hanoi problem
Practical	Tic Tac toe

5

Practical

6

Practical

7

Practical

8

Practical 1

Aim:-Implement DFS

```
graph = {'A': set(['B', 'C']),
'B': set(['A', 'D', 'E']),
'C': set(['A', 'F']),
'D': set(['B']),
'E': set(['B', 'F']),
'F': set(['C', 'E'])}
def dfs(graph, start):
visited, stack = [], [start]
while stack:
vertex = stack.pop()
   if vertex not in visited:visited.append(vertex)
stack.extend(graph[vertex] - set(visited))
return visited

dfs(graph, 'A')
```

Output:

```
['A', 'B', 'D', 'E', 'F', 'C']
```

Practical 2

Aim : Implement BFS

Code:

```
graph = {'A': set(['B', 'C']),
                'B': set(['A', 'D', 'E']),
'C': set(['A', 'F']),
              'D': set(['B']),
'E': set(['B', 'F']),
'F': set(['C', 'E'])}
print(graph);

# def bfs(graph, start):
#   visited,queue=[],[start]    #
whilequeue:
```

```
#    vertex = queue.pop(0)
#    if vertex not in visited:
#    visited.append(vertex)
#     queue.extend(graph[vertex] - set(visited))
#    return visited

# bfs(graph, 'A')

# def bfs_paths(graph, start, goal):
# queue=[(start,[start])]
# whilequeue:
# (vertex,path)=queue.pop(0)
# fornextingraph[vertex]-set(path):
# ifnext==goal:
# yieldpath+[next]
# else:
# queue.append((next,path+[next]))

# list(bfs_paths(graph, 'A', 'F'))
```

Output:

```
{'A': {'C', 'B'}, 'B': {'A', 'E', 'D'}, 'C': {'A', 'F'}, 'D': {'B'}, 'E': {'B', 'F'}, 'F':
{'C', 'E'}}
```

```
# graph = {'A': set(['B', 'C']),
#          'B':set(['A','D','E']),          #
'C':set(['A','F']),
# 'D':set(['B']),
# 'E':set(['B','F']),
# 'F':set(['C','E'])}
# print(graph);

def bfs(graph, start):
  visited, queue = [], [start]
while queue:
vertex = queue.pop(0)
```

```
  if vertex not in visited:
  visited.append(vertex)
     queue.extend(graph[vertex] - set(visited))
  return visited

bfs(graph, 'A')

# def bfs_paths(graph, start, goal):
# queue=[(start,[start])]
# whilequeue:
# (vertex,path)=queue.pop(0)
# fornextingraph[vertex]-set(path):
# ifnext==goal:
# yieldpath+[next]
# else:
# queue.append((next,path+[next]))

# list(bfs_paths(graph, 'A', 'F'))
```

Output:

```
['A', 'C', 'B', 'F', 'E', 'D']
```

```
# graph = {'A': set(['B', 'C']),
#           'B':set(['A','D','E']),          #
'C':set(['A','F']),
# 'D':set(['B']),
# 'E':set(['B','F']),
# 'F':set(['C','E'])}
# print(graph);

# def bfs(graph, start):
# visited,queue=[],[start]
# whilequeue:
# vertex=queue.pop(0)
# ifvertexnotinvisited:
# visited.append(vertex)
#   queue.extend(graph[vertex]-set(visited))   #
returnvisited
```

```
# bfs(graph, 'A')

def bfs_paths(graph, start, goal):
queue = [(start, [start])]
while queue:
(vertex, path) = queue.pop(0)
for next in graph[vertex] - set(path):
if next == goal:
yield path + [next]
else:
            queue.append((next, path + [next]))

list(bfs_paths(graph, 'A', 'F'))
```

Output:

```
[['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
```

Practical 3

Aim: Implement Shortest Path

Code:

```
def bfs_paths(graph, start, goal):
queue = [(start, [start])]
while queue:
(vertex, path) = queue.pop(0)
for next in graph[vertex] - set(path):
if next == goal:
yield path + [next]
else:
                    queue.append((next, path + [next]))

list(bfs_paths(graph, 'A', 'F'))
```

Output:

```
[['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
def bfs_paths(graph, start, goal):
queue = [(start, [start])]
while queue:
(vertex, path) = queue.pop(0)
for next in graph[vertex] - set(path):
if next == goal:
yield path + [next]
else:
        queue.append((next, path + [next]))

list(bfs_paths(graph, 'A', 'F'))

# ShortestPath

def shortest_path(graph, start, goal):
try:
  return next(bfs_paths(graph, start, goal))
except StopIteration:
```

```
    return None

shortest_path(graph, 'A', 'F')
```

Output:

```
['A', 'C', 'F']
```

Practical 4

Aim: . N-queen problem

Code:

```
global N
N=5

def printSolution(board):
for i in range(N):
for j in range(N):
                    print(board[i][j], end=" ")
print()

def isSafe(board, row, col):
for i in range(col):
               if board[row][i] == 1:
return False

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
    if board[i][j] == 1:
    return False

    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
    if board[i][j] == 1:
    return False

    return True

def solveNQUtil(board, col):
```

```
    if col >= N:
        return True

    for i in range(N):
    if isSafe(board, i, col):
    board[i][col] = 1
                if solveNQUtil(board, col + 1):
    return True
    board[i][col] = 0
    return False

def solveNQ():
board = [
            [0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]
    ]

    if not solveNQUtil(board, 0):
        print("Solution does not exist")
    return False

    printSolution(boar
    d) return True

solveNQ()
```

Output:

```
1 0 0 0 0
0 0 0 1 0
0 1 0 0 0
0 0 0 0 1
0 0 1 0 0
True
```

Practical 5

Aim : hanoi problem

Code:

```
def tower_of_hanoi(n, source_peg, auxiliary_peg, target_peg):
if n == 1:
        print(f"Move disk 1 from {source_peg} to {target_peg}")
return
tower_of_hanoi(n-1, source_peg, target_peg, auxiliary_peg)
print(f"Move disk {n} from {source_peg} to {target_peg}")
tower_of_hanoi(n-1, auxiliary_peg, source_peg, target_peg)

# Example usage
n = 3 # Number of disks
tower_of_hanoi(n, 'A', 'B', 'C')
```

Output:

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

Practical 6

Aim: Jug water problem Code:

```
#Practical 6

capacity = (12, 8, 5) x
= capacity[0]
y = capacity[1]
z = capacity[2]
memory = {}
ans = []

def get_all_states(state):
a = state[0]
b = state[1]
c = state[2]

    if (a == 6 and b == 6):
    ans.append(state)
    return True

    if (a, b, c) in memory:
    return False

    memory[(a, b, c)] = 1

    ifa>0:
    ifa+b<=y:
    if get_all_states((0, a + b, c)):
    ans.append(state)
    return True
    else:
                    if get_all_states((a - (y - b), y, c)):
    ans.append(state)
    return True

    ifa+c<=z:
            if get_all_states((0, b, a + c)):
    ans.append(state)
```

```
      return True
      else:
                    if get_all_states((a - (z - c), b, z)):
      ans.append(state)
      return True

ifb>0:
ifa+b<=x:
if get_all_states((a + b, 0, c)):
ans.append(state)
return True
else:
                    if get_all_states((x, b - (x - a), c)):
ans.append(state)
return True

ifb+c<=z:
if get_all_states((a, 0, b + c)):
ans.append(state)
return True
else:
                    if get_all_states((a, b - (z - c), z)):
ans.append(state)
return True

ifc>0:
ifa+c<=x:
if get_all_states((a + c, b, 0)):
ans.append(state)
return True
else:
                    if get_all_states((x, b, c - (x - a))):
ans.append(state)
return True

      ifb+c<=y:
                if get_all_states((a, b + c, 0)):
      ans.append(state)
      return True
      else:
```

```
            if get_all_states((a, y, c - (y - b))):
            ans.append(state)
            return True

    return False

initial_state = (12, 0, 0)
print("Starting work...\n")
get_all_states(initial_state)
ans.reverse()
for i in ans:
print(i)
```

Output:

```
Starting work...

(12, 0, 0)
(4, 8, 0)
(0, 8, 4)
(8, 0, 4)
(8, 4, 0)
(3, 4, 5)
(3, 8, 1)
(11, 0, 1)
(11, 1, 0)
(6, 1, 5)
(6, 6, 0)
```

Practical 7

Aim: Missionaries and Cannibals

Code:

```
class State():
      def __init__(self, cannibalLeft, missionaryLeft, boat, cannibalRight,
missionaryRight):
self.cannibalLeft = cannibalLeft
self.missionaryLeft = missionaryLeft
self.boat = boat
self.cannibalRight = cannibalRight
self.missionaryRight = missionaryRight
self.parent = None

    def is_goal(self):
            if self.cannibalLeft == 0 and self.missionaryLeft == 0:
    return True
    else:
    return False

    def is_valid(self):
    if(
    self.missionaryLeft >= 0
    and self.missionaryRight >= 0
    and self.cannibalLeft >= 0
    and self.cannibalRight >= 0
    and(
    self.missionaryLeft == 0
    or self.missionaryLeft >= self.cannibalLeft
    )
    and(
    self.missionaryRight == 0
                 or self.missionaryRight >= self.cannibalRight
            )
        ):
        return True
        else:
            return False

    def __eq__(self, other):
```

```
        return (
        self.cannibalLeft == other.cannibalLeft
        and self.missionaryLeft == other.missionaryLeft
        and self.boat == other.boat
            and self.cannibalRight == other.cannibalRight
          and self.missionaryRight == other.missionaryRight
        )

    def __hash__(self):
    return hash(
    (
    self.cannibalLeft,
    self.missionaryLeft,
    self.boat,
    self.cannibalRight,
                self.missionaryRight,
            )
        )

def successors(cur_state):
children = []
if cur_state.boat == "left":
new_state = State(
cur_state.cannibalLeft,
cur_state.missionaryLeft - 2,
"right",
cur_state.cannibalRight,
            cur_state.missionaryRight + 2,
)
if new_state.is_valid():
new_state.parent = cur_state
children.append(new_state)


        new_state = State(
        cur_state.cannibalLeft - 2,
        cur_state.missionaryLeft,
        "right",
            cur_state.cannibalRight + 2,
        cur_state.missionaryRight,
        )
```

```
      if new_state.is_valid():
        new_state.parent = cur_state
      children.append(new_state)

      new_state = State(
      cur_state.cannibalLeft - 1,
      cur_state.missionaryLeft - 1,
      "right",
      cur_state.cannibalRight + 1,
         cur_state.missionaryRight + 1,
      )
      if new_state.is_valid():
      new_state.parent = cur_state
        children.append(new_state)

      new_state = State(
      cur_state.cannibalLeft,
      cur_state.missionaryLeft - 1,
      "right",
      cur_state.cannibalRight,
         cur_state.missionaryRight + 1,
      )
      if new_state.is_valid():
      new_state.parent = cur_state
        children.append(new_state)

   new_state = State(
   cur_state.cannibalLeft - 1,
   cur_state.missionaryLeft,
   "right",
   cur_state.cannibalRight + 1,
   cur_state.missionaryRight,
   )
   if new_state.is_valid():
   new_state.parent = cur_state
   children.append(new_state)
   else:
   new_state = State(
   cur_state.cannibalLeft,
         cur_state.missionaryLeft + 2,
```

```
"left",
cur_state.cannibalRight,
     cur_state.missionaryRight - 2,
)
if new_state.is_valid():
new_state.parent = cur_state
   children.append(new_state)

new_state = State(
cur_state.cannibalLeft + 2,
cur_state.missionaryLeft,
"left",
     cur_state.cannibalRight - 2,
cur_state.missionaryRight,
)
if new_state.is_valid():
   new_state.parent = cur_state
children.append(new_state)

new_state = State(
cur_state.cannibalLeft + 1,
cur_state.missionaryLeft + 1,
"left",
cur_state.cannibalRight - 1,
     cur_state.missionaryRight - 1,
)
if new_state.is_valid():
new_state.parent = cur_state
   children.append(new_state)

new_state = State(
cur_state.cannibalLeft,
cur_state.missionaryLeft + 1,
"left",
cur_state.cannibalRight,
     cur_state.missionaryRight - 1,
)
if new_state.is_valid():
new_state.parent = cur_state
   children.append(new_state)
```

```
            new_state = State(
            cur_state.cannibalLeft + 1,
            cur_state.missionaryLeft,
            "left",
                   cur_state.cannibalRight - 1,
            cur_state.missionaryRight,
            )
            if new_state.is_valid():
                new_state.parent = cur_state
            children.append(new_state)

      return children

def breadth_first_search():
initial_state = State(3, 3, "left", 0, 0)
if initial_state.is_goal():
return initial_state
frontier = list()
explored = set()
frontier.append(initial_state)
while frontier:
state = frontier.pop(0)
if state.is_goal():
return state
explored.add(state)
children = successors(state)
for child in children:
                         if (child not in explored) or (child not in frontier):
frontier.append(child)
return None


def print_solution(solution):
path = []
path.append(solution)
parent = solution.parent
while parent:
path.append(parent)
          parent = parent.parent
```

```
    for t in range(len(path)):
    state = path[len(path) - t - 1]
    print(
    "("
    + str(state.cannibalLeft)
    +","
    + str(state.missionaryLeft)
    +","
    + state.boat
    +","
    + str(state.cannibalRight)
    +","
            + str(state.missionaryRight)
    +")"
        )

def main():
solution = breadth_first_search()
    print("Missionaries and Cannibals solution:")

print("(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)")
print_solution(solution)

#  if  called  from  the  command  line,  call
main() if __name__ == "__main__":
main()
```

Output:

```
Missionaries and Cannibals solution:
(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)
(3,3,left,0,0)
(1,3,right,2,0)
(2,3,left,1,0)
(0,3,right,3,0)
(1,3,left,2,0)
(1,1,right,2,2)
(2,2,left,1,1)
(2,0,right,1,3)
(3,0,left,0,3)
(1,0,right,2,3)
(1,1,left,2,2)
(0,0,right,3,3)
```

Practical 8

Aim: Hanoiproblem

Code:

```
#Practical 8

def hanoi(n, P1, P2, P3):
        """ Move n discs from pole P1 to pole P3. """
if n == 0:
# No more discs to move in this step
return

     global
     count
     count += 1
     # move n-1 discs from P1 to
     P2 hanoi(n - 1, P1, P3, P2)

     if P1:
     # move disc from P1 to P3
     P3.append(P1.pop())
          print(A, B, C) # Print the current state of the poles

     # move n-1 discs from P2 to
     P3 hanoi(n - 1, P2, P1, P3)

# Initialize the poles: all n discs are on pole A.
n=5
A = list(range(n, 0, -1))
B, C = [], []

print(A, B, C)

count = 0
hanoi(n, A, B, C)
print(count)
```

Output:

```
[5, 4, 3, 2, 1] [] []
[5, 4, 3, 2] [] [1]
[5, 4, 3] [2] [1]
[5, 4, 3] [2, 1] []
[5, 4] [2, 1] [3]
[5, 4, 1] [2] [3]
[5, 4, 1] [] [3, 2]
[5, 4] [] [3, 2, 1]
[5] [4] [3, 2, 1]
[5] [4, 1] [3, 2]
[5, 2] [4, 1] [3]
[5, 2, 1] [4] [3]
[5, 2, 1] [4, 3] []
[5, 2] [4, 3] [1]
[5] [4, 3, 2] [1]
[5] [4, 3, 2, 1] []
[] [4, 3, 2, 1] [5]
[1] [4, 3, 2] [5]
[1] [4, 3] [5, 2]
[] [4, 3] [5, 2, 1]
[3] [4] [5, 2, 1]
[3] [4, 1] [5, 2]
[3, 2] [4, 1] [5]
[3, 2, 1] [4] [5]
```

Practical 9

Aim : Tic Tac

Toe Code:

```
def print_board(board):
for row in board:
             print(" | ".join(row))
           print("-" * 9)

def check_winner(board, player):
for row in board:
if all(cell == player for cell in row):
return True
for col in range(3):
if all(board[row][col] == player for row in range(3)):
return True
                if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i]
== player for i in range(3)):
```

```
        return True
    return False

def is_full(board):
         return all(cell != " " for row in board for cell in row)

def main():
        board = [[" " for _ in range(3)] for _ in range(3)]
current_player = "X"

    print("Welcome    to    Tic-Tac-
    Toe!") print_board(board)

while True:
row = int(input(f"Player {current_player}, enter the row (0, 1,
2): "))
           col = int(input(f"Player {current_player}, enter the column (0, 1,
2): "))

        if board[row][col] == " ":
        board[row][col] = current_player
        else:
             print("Invalid move. Try again.")
        continue

        print_board(board)

        if check_winner(board, current_player):
             print(f"Player {current_player} wins! Congratulations!")
        break

        if is_full(board):
             print("It's a draw! No winner.")
        break

        current_player = "O" if current_player == "X" else "X"

if __name__ == "__main__":
main()
```

Output:

```
Welcome to Tic-Tac-Toe!
  |   |
---------
  |   |
---------
  |   |
---------
Player X, enter the row (0, 1, 2): 0
Player X, enter the column (0, 1, 2): 1
  | X |
---------
  |   |
---------
  |   |
---------
Player O, enter the row (0, 1, 2):
```