B-Tech III Year II Semester (3-2)

CSE

Artificial Intelligence

Lab Manual

Dr. Seva Sreedhar Babu
D.E.C.E.,B.Tech.,M.Tech.,Ph.D.,M.I.S.T.E.,A.M.I.A.E
ASSOC.PROF

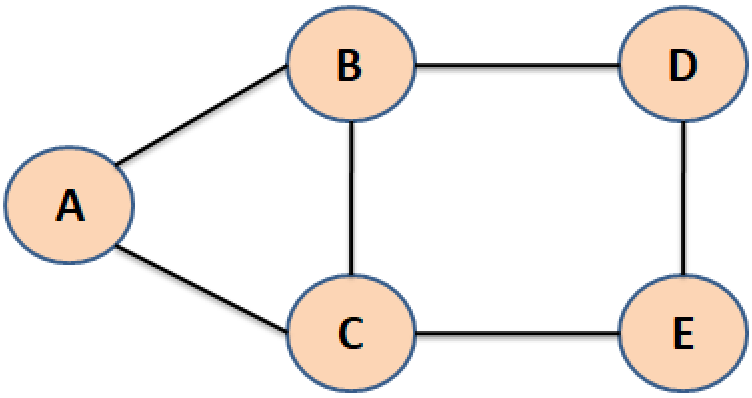
SREE DATTHA INSTITUTIONS

Engineering | Pharmacy | Management | Polytechnic | B.Ed

Programs to Implement the following using Python	
1.	**Breadth First Search**
2.	**Depth First Search**
3.	**Tic-Tac-Toe game**
4.	**8-Puzzle problem**
5.	**Water-Jug problem**
6.	**Travelling Salesman Problem**
7.	**Tower of Hanoi**
8.	**Monkey Banana Problem**
9.	**Alpha-Beta Pruning**
10.	**8-Queens Problem**

1.Write a Program to implement Breadth First Search (BFS) using Python.

Input Graph



SOURCE CODE :

```
 # Input Graph
graph = {'A' : ['B','C'],
'B' : ['A','C','D'],
'C' : ['A','B','E'],
'D' : ['B','E'],
'E' : ['C','D']
}
# To store visited nodes.
visitedNodes = []
# To store nodes in queue
queueNodes = []
 # function
def bfs(visitedNodes, graph, snode):
       visitedNodes.append(snode)
```

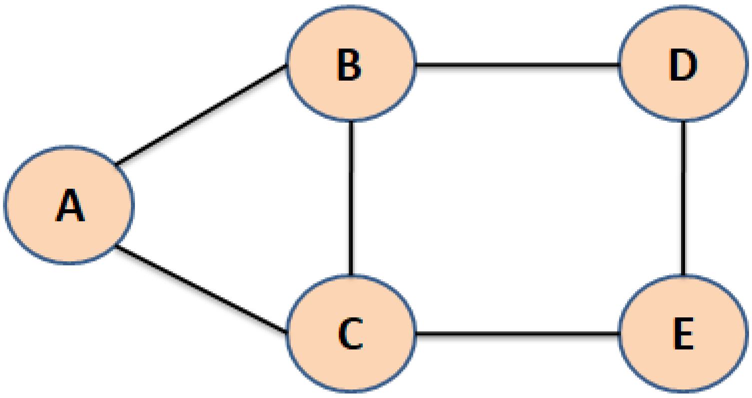
```python
        queueNodes.append(snode)
        print()
        print("RESULT :")
        while queueNodes:
                s = queueNodes.pop(0)
                print (s, end = " ")
                for neighbour in graph[s]:
                        if neighbour not in visitedNodes:
                                visitedNodes.append(neighbour)
                                queueNodes.append(neighbour)
# Main Code
snode = input("Enter Starting Node(A, B, C, D, or E) :").upper
()
# calling bfs function
bfs(visitedNodes, graph, snode)
```

OUTPUT :

```
Sample Output 1:
-------------------
Enter Starting Node(A, B, C, D, or E) :A

RESULT :
A B C D E
------------------------------------------------------------------
Sample Output 2:
-------------------
Enter Starting Node(A, B, C, D, or E) :B

RESULT :    B A C D E
```

1. Write a Program to implement Depth First Search (DFS) using Python.

Input Graph



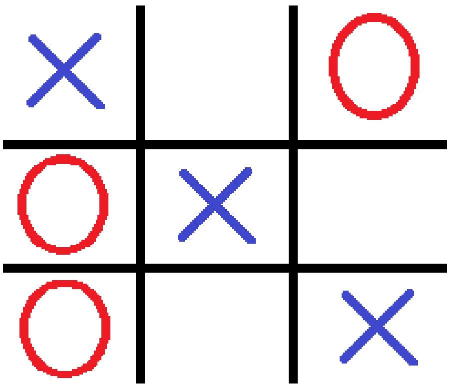
SOURCE CODE :

```
# Input Graph
graph = {
'A' : ['B','C'],
'B' : ['A','C','D'],
'C' : ['A','B','E'],
'D' : ['B','E'],
'E' : ['C','D']
}
# Set used to store visited nodes.
visitedNodes = list()
# function
def dfs(visitedNodes, graph, node):
        if node not in visitedNodes:
                print (node,end=" ")
                visitedNodes.append(node)
                for neighbour in graph[node]:
                        dfs(visitedNodes, graph, neighbour)
# Driver Code
snode = input("Enter Starting Node(A, B, C, D, or E) :").upper
()
# calling bfs function
print("RESULT :")
print("-"*20)
dfs(visitedNodes, graph, snode)
```

OUTPUT :

```
Sample Output 1:

Enter Starting Node(A, B, C, D, or E) :A

RESULT : A B C E D

-----------------------------------------------------------------

Sample Output 2:

Enter Starting Node(A, B, C, D, or E) :B

RESULT :B A C E D
```

2. Write a Program to implement Tic-tac-toe game using Python.



SOURCE CODE :

```python
 # Tuple to store winning positions.
win_positions = (
    (0, 1, 2), (3, 4, 5), (6, 7, 8),
    (0, 3, 6), (1, 4, 7), (2, 5, 8),
    (0, 4, 8), (2, 4, 6)
    )
```

```
 def game(player):
    # diplay current mesh
    print("\n", " | ".join(mesh[:3]))
    print("---+---+---")
    print("", " | ".join(mesh[3:6]))
    print("---+---+---")
    print("", " | ".join(mesh[6:]))

    # Loop until player valid input cell number.
    while True:
        try:
            ch = int(input(f"Enter player {player}'s choice :
"))
            if str(ch) not in mesh:
                raise ValueError
            mesh[ch-1] = player
            break
        except ValueError:
            print("Invalid position number.")

    # Return winning positions if player wins, else None.
    for wp in win_positions:
        if all(mesh[pos] == player for pos in wp):
            return wp
    return None

player1 = "X"
player2 = "O"
player = player1
mesh = list("123456789")
 for i in range(9):
       won = game(player)
       if won:
              print("\n", " | ".join(mesh[:3]))
              print("---+---+---")
              print("", " | ".join(mesh[3:6]))
              print("---+---+---")
              print("", " | ".join(mesh[6:]))
              print(f"*** Player {player} won! ***")
              break
       player = player1 if player == player2 else player2
else:
    # 9 moves without a win is a draw.
    print("Game ends in a draw.")
```

OUTPUT :

```
Sample Output:

1 | 2 | 3

---+---+---

4 | 5 | 6

---+---+---

7 | 8 | 9

Enter player X's choice : 5

1 | 2 | 3

---+---+---

4 | X | 6

---+---+---

7 | 8 | 9

Enter player O's choice : 3

1 | 2 | O

---+---+---

4 | X | 6

---+---+---

7 | 8 | 9

Enter player X's choice : 1

X | 2 | O
```

```
---+---+---

4 | X | 6

---+---+---

7 | 8 | 9

Enter player O's choice : 6

X | 2 | O

---+---+---

4 | X | O

---+---+---

7 | 8 | 9

Enter player X's choice : 9

X | 2 | O

---+---+---

4 | X | O

---+---+---

7 | 8 | X

*** Player X won! ***
```

3. Write a Program to Implement 8-Puzzle problem using Python.

SOURCE CODE :

```python
from collections import deque
def bfs(start_state):
    target = [1, 2, 3, 4, 5, 6, 7, 8 , 0]
    dq = deque([start_state])
    visited = {tuple(start_state): None}

    while dq:
        state = dq.popleft()
        if state == target:
            path = []
            while state:
                path.append(state)
                state = visited[tuple(state)]
            return path[::-1]

        zero = state.index(0)
        row, col = divmod(zero, 3)
        for move in (-3, 3, -1, 1):
            new_row, new_col = divmod(zero + move, 3)
            if 0 <= new_row < 3 and 0 <= new_col < 3 and abs(row - new_row) + abs(col - new_col) == 1:
                neighbor = state[:]
```

```
  neighbor[zero], neighbor[zero + move] = neighbor[zero + mov
e], neighbor[zero]

                if tuple(neighbor) not in visited:

                    visited[tuple(neighbor)] = state

                    dq.append(neighbor)

def printSolution(path):

    for state in path:

        print("\n".join(' '.join(map(str, state[i:i+3])) for i
 in range(0, 9, 3)), end="\n-----\n")

# Example Usage

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

solution = bfs(startState)

if solution:

    printSolution(solution)

    print(f"Solved in {len(solution) - 1} moves.")

    else:

    print("No solution found.")
```

OUTPUT :

```
1 3 0

6 8 4

7 5 2

-----

1 3 4

6 8 0

7 5 2
```

```
-----

1 3 4

6 8 2

7 5 0

-----

1 3 4

6 8 2

7 0 5

-----

.

.

.

-----

1 2 3

4 5 0

7 8 6

-----

1 2 3

4 5 6

7 8 0

-----

Solved in 20 moves.
```

4. Write a Program to Implement Water-Jug problem using Python..

SOURCE CODE :

```
 # jug1 and jug2 contain the value
jug1, jug2, goal = 4, 3, 2
# Initialize a 2D list for visited states# The list will have dimensions (jug1+1) x (jug2+1) to cover all possible states
visited = [[False for _ in range(jug2 + 1)] for _ in range(jug1 + 1)]
def waterJug(vol1, vol2):
       # Check if we reached the goal state
       if (vol1 == goal and vol2 == 0) or (vol2 == goal and vol1 == 0):
              print(vol1,"\t", vol2)
              print("Solution Found")
              return True


    # If this state has been visited, return False
       if visited[vol1][vol2]:
              return False
       # Mark this state as visited
       visited[vol1][vol2] = True
       # Print the current state
       print(vol1,"\t", vol2)
       # Try all possible moves:
       return (
```

```
        waterJug(0, vol2) or  # Empty jug1
        waterJug(vol1, 0) or  # Empty jug2
        waterJug(jug1, vol2) or  # Fill jug1
        waterJug(vol1, jug2) or  # Fill jug2
        waterJug(vol1 + min(vol2, (jug1 - vol1)), vol2 - min(v
ol2, (jug1 - vol1))) or  # Pour water from jug2 to jug1
        waterJug(vol1 - min(vol1, (jug2 - vol2)), vol2 + min(v
ol1, (jug2 - vol2)))  # Pour water from jug1 to jug2
    )
print("Steps: ")
print("Jug1 \t Jug2 ")
print("----- \t ------")
waterJug(0, 0)
```

OUTPUT : Steps:

```
Jug1    Jug2
-----   ------
0       0
4       0
4       3
0       3
3       0
3       3
4       2
0       2
Solution Found
```

5. Write a Program to Implement Travelling Salesman Problem using Python.

SOURCE CODE :

```
 from collections import deque
def tsp_bfs(graph):
    n = len(graph)  # Number of cities
    startCity = 0       # Starting city
    min_cost = float('inf')  # Initialize minimum cost as infinity
    opt_path = []        # To store the optimal path

    # Queue for BFS: Each element is (cur_path, cur_cost)
    dq = deque([([startCity], 0)])

    print("Path Traversal:")

    while dq:
        cur_path, cur_cost = dq.popleft()
        cur_city = cur_path[-1]

        # Print the current path and cost
        print(f"Current Path: {cur_path}, Current Cost: {cur_cost}")

    If all cities are visited and we are back at the startCity
        if len(cur_path) == n and cur_path[0] == startCity:
            total_cost = cur_cost + graph[cur_city][startCity]
```

```
            if total_cost < min_cost:
                min_cost = total_cost
                opt_path = cur_path + [startCity]
            continue

        # Explore all neighboring cities (add in BFS manner)
        for next_city in range(n):
        if next_city not in cur_path:  # Visit unvisited cities
                new_path = cur_path + [next_city]
               new_cost = cur_cost + graph[cur_city][next_cit
y]
                dq.append((new_path, new_cost))

    return min_cost, opt_path
# Example graph as a 2D adjacency matrix
graph = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]]
# Solve TSP using BFS
min_cost, opt_path = tsp_bfs(graph)
print("\nOptimal Solution:")
print(f"Minimum cost: {min_cost}")
print(f"Optimal path: {opt_path}")
```

6. Write a Program to Implement Tower of Hanoi using Python.

```
def tower_of_hanoi(num, source, aux, target):
    """
       num (int): Number of disks.
    source (str): The name of the source tower.
    aux (str): The name of the auxiliary tower.
    target (str): The name of the target tower.
    """
    if num == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    # Move num-1 disks from source to auxiliary
    tower_of_hanoi(num - 1, source, target, aux)
    print(f"Move disk {num} from {source} to {target}")
    # Move the num-1 disks from auxiliary to target
    tower_of_hanoi(num - 1, aux, source, target)
# Example usage
num_disks = 3
tower_of_hanoi(num_disks, "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
```

7. Write a Program to Implement Monkey Banana Problem using Python.

SOURCE CODE :

```python
def monkey_banana_problem():
    # Initial state
    initial_state = ('Far-Chair', 'Chair-Not-Under-Banana', 'Off-Chair', 'Empty')  # (Monkey's Location, Monkey's Position on Chair, Chair's Location, Monkey's Status)
    print(f"\n Initial state is {initial_state}")
    goal_state = ('Near-Chair', 'Chair-Under-Banana', 'On-Chair', 'Holding')     # The goal state when the monkey has the banana


    # Possible actions and their effects
    actions = {
        "Move to Chair": lambda state: ('Near-Chair', state[1], state[2], state[3]) if state[0] != 'Near-Chair' else None,
        "Push Chair under Banana": lambda state: ('Near-Chair', 'Chair-Under-Banana', state[2],  state[3]) if state[0] == 'Near-Chair' and state[1] != 'Chair-Under-Banana' else None,
        "Climb Chair": lambda state: ('Near-Chair', 'Chair-Under-Banana', 'On-Chair', state[3]) if state[0] == 'Near-Chair' and state[1] == 'Chair-Under-Banana' and state[2] != 'On-Chair' else None,
        "Grasp Banana": lambda state: ('Near-Chair', 'Chair-Under-Banana', 'On-Chair',  'Holding') if state[0] == 'Near-Chair' and state[1] == 'Chair-Under-Banana' and state[2] == 'On-Chair' and state[3] !='Holding' else None
```

```
    }


    # BFS to explore states
    from collections import deque
    dq = deque([(initial_state, [])])  # Each element is (curr
ent_state, actions_taken)
    visited = set()


    while dq:
        current_state, actions_taken = dq.popleft()


        # Check if we've reached the goal
        if current_state == goal_state:
            print("\nSolution Found!")
            print("Actions to achieve goal:")
            for action in actions_taken:
                print(action)
            print(f"Final State: {current_state}")
            return


        # Mark the current state as visited
        if current_state in visited:
            continue
        visited.add(current_state)


        # Try all possible actions
        for action_name, action_func in actions.items():
            next_state = action_func(current_state)
```

```python
            if next_state and (next_state not in visited):
                dq.append((next_state, actions_taken + [f"Action: {action_name}, Resulting State: {next_state}"]))


    print("No solution found.")

# Run the program

monkey_banana_problem()
```

OUTPUT :

```
Initial state is ('Far-Chair', 'Chair-Not-Under-Banana', 'Off-Chair', 'Empty')



Solution Found!

Actions to achieve goal:

Action: Move to Chair, Resulting State: ('Near-Chair', 'Chair-Not-Under-Banana', 'Off-Chair', 'Empty')

Action: Push Chair under Banana, Resulting State: ('Near-Chair', 'Chair-Under-Banana', 'Off-Chair', 'Empty')

Action: Climb Chair, Resulting State: ('Near-Chair', 'Chair-Under-Banana', 'On-Chair', 'Empty')

Action: Grasp Banana, Resulting State: ('Near-Chair', 'Chair-Under-Banana', 'On-Chair', 'Holding')

Final State: ('Near-Chair', 'Chair-Under-Banana', 'On-Chair', 'Holding')
```

9.Write a Program to Implement Alpha-Beta Pruning using Python.

SOURCE CODE :

```
 """

      Alpha Beta Pruning :

      -------------------

    depth (int): Current depth in the game tree.

    node_index (int): Index of the current node in the values
array.

    maximizing_player (bool): True if the current player is ma
ximizing, False otherwise.

    values (list): List of leaf node values.

    alpha (float): Best value for the maximizing player.

    beta (float): Best value for the minimizing player.



    Returns:

    int: The optimal value for the current player.

    """import math

def alpha_beta_pruning(depth, node_index, maximizing_player, v
alues, alpha, beta):

    # Base case: leaf node

    if depth == 0 or node_index >= len(values):

        return values[node_index]
```

```
    if maximizing_player:
        max_eval = -math.inf
        for i in range(2):  # Each node has two children
            eval = alpha_beta_pruning(depth - 1, node_index *
2 + i, False, values, alpha, beta)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Beta cutoff
        return max_eval
    else:
        min_eval = math.inf
        for i in range(2):  # Each node has two children
            eval = alpha_beta_pruning(depth - 1, node_index *
2 + i, True, values, alpha, beta)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  # Alpha cutoff
        return min_eval
# Example usageif __name__ == "__main__":
    # Leaf node values for a complete binary tree
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    depth = 3  # Height of the tree
    optimal_value = alpha_beta_pruning(depth, 0, True, values,
 -math.inf, math.inf)
    print(f"The optimal value is: {optimal_value}")
```

OUTPUT :

```
The optimal value is: 5
```

8. Write a Program to Implement 8-Queens Problem using Python.

SOURCE CODE :

```python
 def printSolution(board):
    """Print the chessboard configuration."""
    for row in board:
        print(" ".join("Q" if col else "." for col in row))
    print("\n")
def isSafe(board, row, col, n):
    """Check if placing a queen at board[row][col] is safe."""
    # Check column
    for i in range(row):
        if board[i][col]:
            return False

    # Check upper-left diagonal
    i, j = row, col
    while i >= 0 and j >= 0:
        if board[i][j]:
            return False
```

```
        i -= 1
        j -= 1

    # Check upper-right diagonal
    i, j = row, col
    while i >= 0 and j < n:
        if board[i][j]:
            return False
        i -= 1
        j += 1

    return True

def solveNQueens(board, row, n):
    """Use backtracking to solve the N-Queens problem."""
    if row == n:
        printSolution(board)
        return True

    result = False
    for col in range(n):
        if isSafe(board, row, col, n):
            # Place the queen
            board[row][col] = 1
            # Recur to place the rest of the queens
            result = solveNQueens(board, row + 1, n) or result
            # Backtrack
```

```python
            board[row][col] = 0


    return result
def nQueens(n):
    """Driver function to solve the N-Queens problem."""
    board = [[0] * n for _ in range(n)]
    if not solveNQueens(board, 0, n):
        print("No solution exists.")
    else:
        print("Solutions printed above.")
# Solve the 8-Queens problem
nQueens(8)
```

OUTPUT :

```
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .


Q . . . . . . .
```

```
. . . . . Q . .

. . . . . . . Q

. . Q . . . . .

. . . . . . Q .

. . . Q . . . .

. Q . . . . . .

. . . . Q . . .


.

.


. . . . . . . Q

. . . Q . . . .

Q . . . . . . .

. . Q . . . . .

. . . . . Q . .

. Q . . . . . .

. . . . . . Q .

. . . . Q . . .



Solutions printed above.
```