Lokmanya Tilak Jankalyan Shikshan Sanstha's

PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR

(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur University)

DEPARTMENT OF INFORMATION TECHNOLOGY

ARTIFICIAL INTELLIGENCE - LAB MANUAL

Subject code: **25UIT501T**

Academic Year : 2025-2026

Semester: V

KNOW ABOUT YOUR LAB

INDEX

LOKMANYA TILAK JANKALYAN SHIKSHAN SANSTHA'S

PRIYADARSHINI COLLEGE OF ENGINEERING

An Autonomous Institute Affilliated to R.T.M. Nagpur University, Nagpur
Accredited with Grade 'A+' by NAAC
Near C.R.P.F. Campus, Hingna Road, Nagpur - 440 019 (Maharashtra) India
Phone : 07104 - 299648, Fax : 07104-299681
E-mail : principal.pce.ngp@gmail.com • Website: www.pcenagpur.edu.in
AICTE ID No. 1-5435581; DTE CODE No. 4123,
UNIVERSITY CODE No. : 278

Vision

To become one of the India's leading Engineering Institutes in both education and research. We are committed to provide quality and state-of-the-art technical education to our students so that they become Technologically competent and in turn contribute for creating a great society.

Mission

1. Fostering a dynamic learning environment that equips students with Technical expertise, problem-solving skills and a deep commitment to ethical practices.
2. To cultivate a culture of innovation, incubation, research and entrepreneurship that drives technological advancements.
3. To uphold the spirit of mutual excellence while interacting with stake holders of our Institutional ecosystem.
4. Promoting lifelong learning, professional growth and ensuring holistic development of students and the well being of society.

PRIYADARSHINI COLLEGE OF ENGINEERING • NAGPUR • PCE

[illegible]

Principal
Priyadarshini College of Engg.
Nagpur.

PROGRAM EDUCATIONAL OBJECTIVES:

PEOs	Program Educational Objectives Statements
PEO1	Demonstrate strong technical expertise and uphold ethical values to excel in their professional careers.
PEO2	Apply advanced skills and knowledge in Information Technology to address real-world challenges and contribute to industry success.
PEO3	Pursue lifelong learning and engage in research and innovation to address societal needs and advance the field of Information Technology.

PROGRAM OUTCOMES:

Engineering Graduates will Able to:

POs	Program Outcomes
PO1	Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
PO2	Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	Conduct investigations of complex problems: Use research-based knowledge and research methods Including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO6	The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10	Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one"s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

DEPARTMENT OF INFORMATION TECHNOLOGY

PROGRAM SPECIFIC OUTCOMES :

PSOs	Program Specific Outcomes
PSO1	An ability to apply mathematical foundations, algorithmic principles and computer science theory in the modeling and design of software systems of varying complexity.
PSO2	An ability to work with Open Source Software and use off the shelf utilities for program integration.

COURSE OUTCOMES :

Course Outcomes	Statement
CO1	To understand and implement various uninformed and informed search algorithms.
CO2	To apply problem-solving strategies and game-based algorithms.
CO3	To explore and implement constraint satisfaction and optimization problems

CO-PO & PSO MAPPING:

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
CO1	3	3	2	2	2	1	1	1	1	1	1	2	3	2
CO2	3	2	3	2	3	1	1	1	1	2	1	2	3	2
CO3	3	2	3	3	3	2	1	1	2	2	1	3	3	3

List of Experiments:

Sr. No.	Name of Practical
1	Write a Program to Implement Breadth First Search using Python.
2	Write a Program to Implement Depth First Search using Python.
3	Write a Program to Implement Tic-Tac-Toe game using Python.
4	Write a Program to Implement 8-Puzzle problem using Python.
5	Write a Program to Implement Water-Jug problem using Python.
6	Write a Program to Implement Travelling Salesman Problem using Python
7	Write a Program to Implement Tower of Hanoi using Python.
8	Write a Program to Implement Monkey Banana Problem using Python.
9	Write a Program to Implement Alpha-Beta Pruning using Python.
10	Write a Program to Implement 8-Queens Problem using Python.
Content Beyond Syllabus	
11	Mini Project

PRACTICAL NO.1

Aim: Write a Program to Implement Breadth First Search using Python.

Theory:

Breadth First Search (BFS) is an uninformed graph traversal algorithm used to explore nodes and edges of a graph systematically. It explores all the nodes at the present depth level before moving on to nodes at the next level.

Working Principle:

- BFS uses a **queue** data structure (FIFO – First In First Out).
- It begins at the **starting (source) node**, visits all its **neighbours**, then moves to the next level of neighbours.
- A **visited list** (or set) is maintained to avoid revisiting the same node.

Applications of BFS:

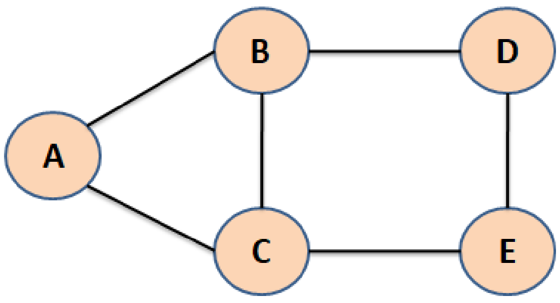
- Finding shortest path in unweighted graphs
- Web crawlers
- Social networking sites (friend suggestion)
- GPS navigation systems
- Network broadcasting

Algorithm :

```
1. Create an empty queue and enqueue the starting node.
2. Mark the starting node as visited.
3. Repeat until the queue is empty:
    o Dequeue the front node from the queue.
    o Process the dequeued node.
    o Enqueue all unvisited adjacent nodes and mark them as visited.
```

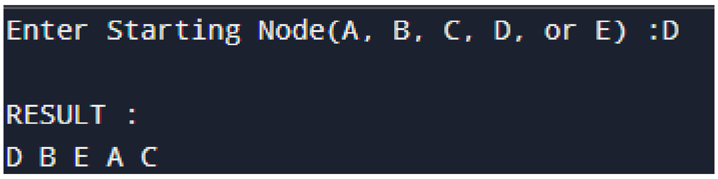
Program:

Input Graph



```
# 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)
        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:



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

RESULT :
D B E A C
```

Conclusion: We have successfully executed Program of Breadth First Search using Python.

Viva Questions and Answers:

1. **What is Breadth First Search (BFS)?**

   Ans:
   BFS is a graph traversal algorithm that explores all nodes at the present depth before going deeper.

2. **What data structure is used in BFS?**

   Ans: A **queue** is used to maintain the order of traversal.

3. **How is BFS different from DFS?**

   Ans: BFS explores **level-wise**; DFS explores **depth-wise** using a **stack or recursion**.

PRACTICAL NO: 2

Aim: Write a Program to Implement Depth First Search using Python.

Theory:

Depth First Search (DFS) is a fundamental **graph traversal** algorithm used in computer science to explore nodes and edges of a graph. It begins at a **starting node (root)** and explores as far as possible along each branch **before backtracking**.

DFS explores a node, marks it as visited, then recursively explores its **unvisited neighbors**.

It is like walking through a **maze**, going as deep as possible before hitting a dead end and turning back.

DFS can be implemented using:

- **Recursion** (via the function call stack), or
- **Explicit stack** (for iterative implementation).

Advantages of DFS:

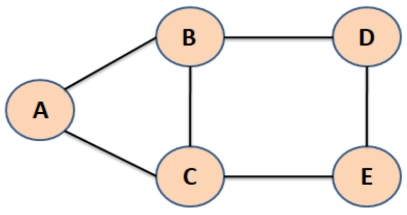
- Requires less memory than BFS (no need to store all neighbors at each level).
- Useful for solving problems where backtracking is needed (like puzzles, topological sorting).

Algorithm: (Recursive DFS)

```
1. Define a function DFS(graph, node, visited):
   a. Mark the current node as visited.
   b. Print or process the current node.
   c. For each neighbor of the current node in graph[node]:
      i. If the neighbor has not been visited:
         - Recursively call DFS(graph, neighbor, visited)

2. Initialize an empty set or list called visited to keep track of visited nodes.
3. Call DFS(graph, start_node, visited)
```

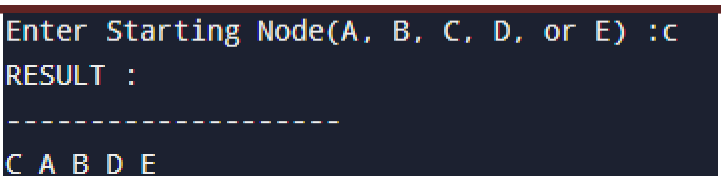
Program:

Input Graph



```
# 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:



```
Enter Starting Node(A, B, C, D, or E) :c
RESULT :
--------------------
C A B D E
```

Conclusion: We have Successfully Executed the Program for Depth First Search using Python.

VIVA QUESTIONS:

1. **What is Depth First Search (DFS)?**
   Ans: DFS is a graph traversal algorithm that starts from a selected node and explores as far as possible along each branch before backtracking.

2. **How is DFS different from BFS?**
   Ans: How is DFS different from BFS?

3. **What are some applications of DFS?**
   Ans: Cycle detection in graphs
   Solving puzzles and mazes
   Topological sorting
   Finding connected components
   Path finding in AI

PRACTICAL NO. 3

Aim: Write a Program to Implement Tic-Tac-Toe game using Python.

Theory:
Tic-Tac-Toe is a simple two-player game played on a 3×3 grid. Each player takes turns marking a space in the grid with their symbol—'X' or 'O'. The first player to get three of their symbols in a row (vertically, horizontally, or diagonally) wins the game. If all 9 squares are filled without a winner, the game ends in a **draw**.
Implementing this game in Python is a common beginner project that strengthens knowledge of:

- Lists (for the game board)
- Conditional logic
- Loops
- Functions
- Input/output handling

Benefits of Implementation:

- Helps understand game loop logic.
- Enhances problem-solving skills through win/draw detection.
- Introduces the concept of turn-based systems and user input validation.

Algorithm:

Step 1: Initialize the Game Board

- Create a 3×3 board using a list of 9 spaces or a 2D list (3x3 matrix).
- Assign symbols 'X' and 'O' to two players.

Step 2: Define Functions

- display_board(board) — Displays the current state of the board.
- check_winner(board, player) — Checks all rows, columns, and diagonals for a winning pattern for the given player.
- check_draw(board) — Checks if the board is full and there's no winner.
- make_move(board, position, player) — Places the player's symbol on the selected position if valid.
- switch_player(current_player) — Alternates between 'X' and 'O'.

Step 3: Start the Game Loop

- Set the current player as 'X'.

Repeat until the game ends:

1. Display the board.
2. Prompt the current player to enter a position (1 to 9).
3. Validate the position:

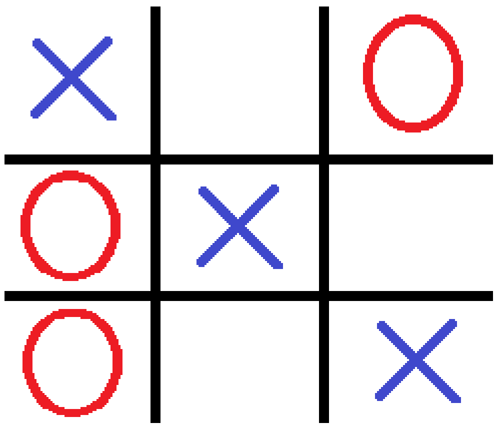
    - If invalid or already occupied, show an error and ask again.
    - If valid, update the board.
4. Check for a winner using check_winner():
    - If true, declare the current player as winner and exit loop.
5. Check for a draw using check_draw():
    - If true, declare the game as a draw and exit loop.
6. Switch to the other player.

Step 4: Display Final Result

- Print the final state of the board.
- Show who won or if it's a draw.

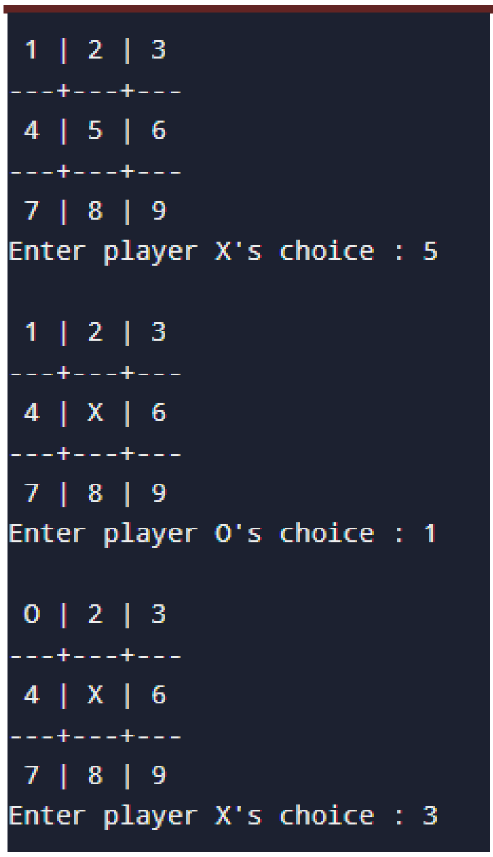
Step 5: End the Program

Program:

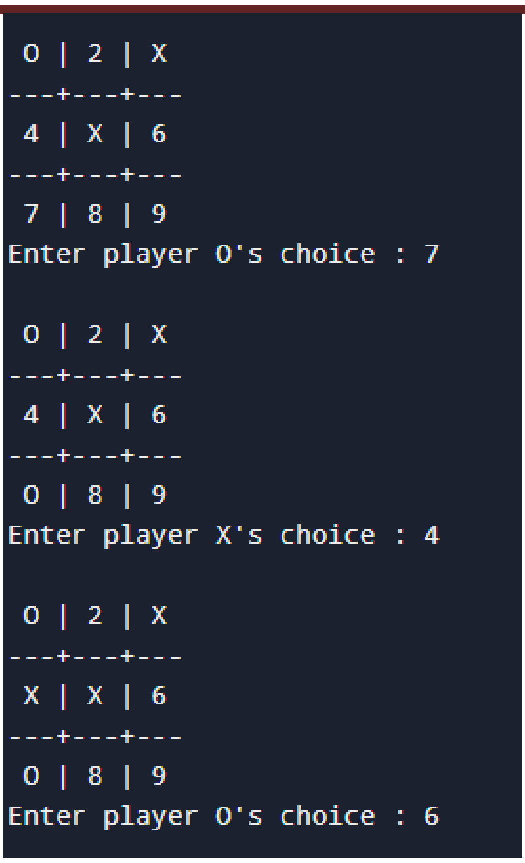


```
# Tuple to store winning positions.
win_positions = (
    (0, 1, 2), (3, 4, 5), (6, 7, 8),
    (0, 3, 6), (1, 4, 7), (2, 5, 8),
    (0, 4, 8), (2, 4, 6)
)

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.")
```

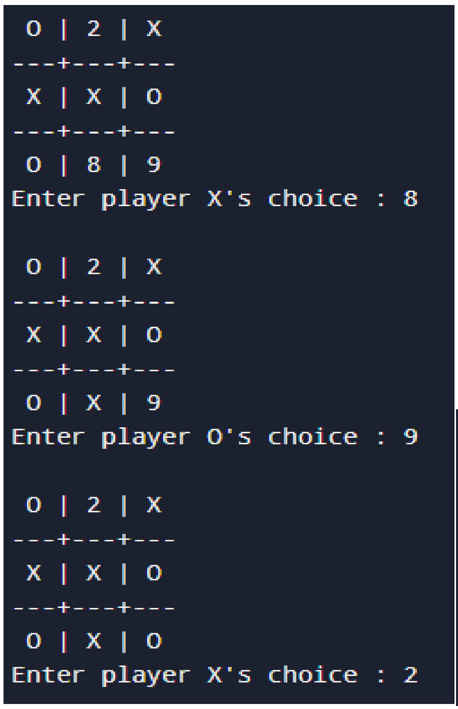
Output:



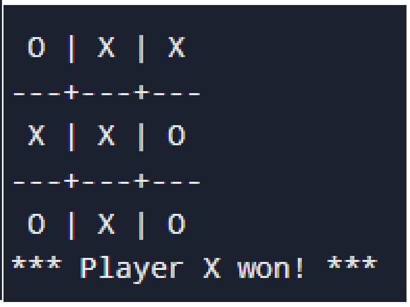
```
 1 | 2 | 3
---+---+---
 4 | 5 | 6
---+---+---
 7 | 8 | 9
Enter player X's choice : 5

 1 | 2 | 3
---+---+---
 4 | X | 6
---+---+---
 7 | 8 | 9
Enter player O's choice : 1

 O | 2 | 3
---+---+---
 4 | X | 6
---+---+---
 7 | 8 | 9
Enter player X's choice : 3
```



```
 O | 2 | X
---+---+---
 4 | X | 6
---+---+---
 7 | 8 | 9
Enter player O's choice : 7

 O | 2 | X
---+---+---
 4 | X | 6
---+---+---
 O | 8 | 9
Enter player X's choice : 4

 O | 2 | X
---+---+---
 X | X | 6
---+---+---
 O | 8 | 9
Enter player O's choice : 6
```



```
 O | 2 | X
---+---+---
 X | X | O
---+---+---
 O | 8 | 9
Enter player X's choice : 8

 O | 2 | X
---+---+---
 X | X | O
---+---+---
 O | X | 9
Enter player O's choice : 9

 O | 2 | X
---+---+---
 X | X | O
---+---+---
 O | X | O
Enter player X's choice : 2
```



```
 O | X | X
---+---+---
 X | X | O
---+---+---
 O | X | O
*** Player X won! ***
```

Conclusion: We have Successfully Executed the Program of Tic-Tac-Toe game using Python.

Viva Question and Answers:

1. **What Python concepts are used in this program?**
   Ans: Lists
   Conditional statements (if-else)
   Loops (while, for)
   Functions
   User input handling

2. **What is the role of the switch_player() function?**
   Ans: It changes the current player from 'X' to 'O' or vice versa after each valid move.

3. **Can this game be extended to single-player mode against the computer?**
   Ans: Yes. An AI component (e.g., using minimax algorithm) can be added for playing against the computer.

PRACTICAL NO. 4

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

Theory:
The 8-puzzle problem is a classic problem in the field of Artificial Intelligence and search algorithms. It consists of a 3x3 grid with 8 numbered tiles (1–8) and one empty space. The goal is to rearrange the tiles from an initial configuration to a goal state by sliding the tiles into the empty space.
The objective is to move the tiles using valid moves (up, down, left, right) until the puzzle reaches the goal configuration. This problem is commonly solved using search algorithms such as:

- Breadth-First Search (BFS)
- A Search Algorithm*
- Depth-First Search (DFS)
- Greedy Best First Search

Applications:

- Understanding heuristic and informed search strategies
- Demonstrates principles of AI planning and pathfinding
- Forms the basis for more complex puzzles (e.g., 15-puzzle, Rubik's cube solvers)

Algorithm:
Step 1: Define the Initial and Goal States

- Represent the puzzle as a 3×3 matrix or a list of 9 numbers.
- The empty space is denoted by 0.

Step 2: Define a Node Structure
Each node contains:

- The current puzzle state.
- The cost to reach this state g(n).
- The heuristic value h(n) (e.g., Manhattan Distance).
- The total cost f(n) = g(n) + h(n).
- The parent node (to trace the path later).

Step 3: Define the Heuristic Function

- Manhattan Distance:

text
CopyEdit
For each tile (except 0), compute:
abs(current_row - goal_row) + abs(current_col - goal_col)

- Sum for all tiles gives h(n).

Step 4: Initialize the Open and Closed Lists
- open_list: Priority queue containing nodes to be explored (sorted by f(n)).
- closed_set: Set of already visited states.

*Step 5: Begin A Search Loop**
1. While open_list is not empty:
   - Remove the node with the lowest f(n) from open_list.
   - If the current state is the goal state:
     - Reconstruct and return the path from start to goal.
   - Add the current state to closed_set.
   - Generate all valid moves (up, down, left, right):
     - Swap the blank tile with the target tile.
     - If the new state is not in closed_set:
       - Calculate g(n) (cost so far).
       - Calculate h(n) using heuristic.
       - Compute f(n) = g(n) + h(n)
       - Add the new node to open_list.

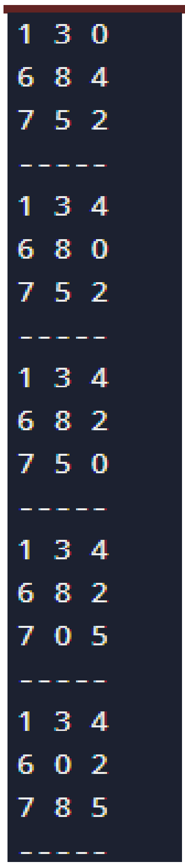
Step 6: If Goal Not Found
- If open_list becomes empty and goal not reached, return failure.

Program:

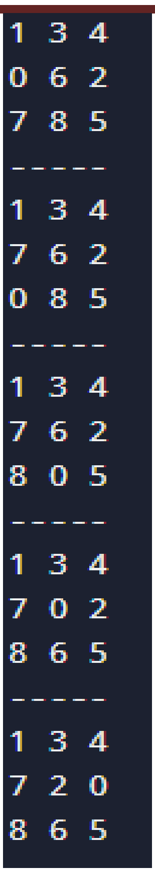
```
from collections import deque

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 + move], 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.")
```

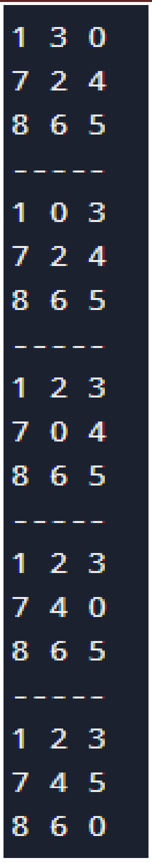
Output:



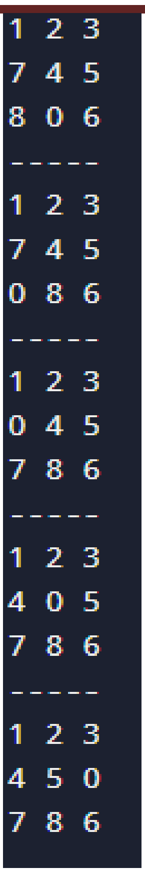
```
1 3 0
6 8 4
7 5 2
-----
1 3 4
6 8 0
7 5 2
-----
1 3 4
6 8 2
7 5 0
-----
1 3 4
6 8 2
7 0 5
-----
1 3 4
6 0 2
7 8 5
-----
```



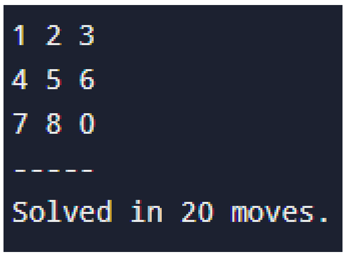
```
1 3 4
0 6 2
7 8 5
-----
1 3 4
7 6 2
0 8 5
-----
1 3 4
7 6 2
8 0 5
-----
1 3 4
7 0 2
8 6 5
-----
1 3 4
7 2 0
8 6 5
```



```
1 3 0
7 2 4
8 6 5
-----
1 0 3
7 2 4
8 6 5
-----
1 2 3
7 0 4
8 6 5
-----
1 2 3
7 4 0
8 6 5
-----
1 2 3
7 4 5
8 6 0
```



```
1 2 3
7 4 5
8 0 6
-----
1 2 3
7 4 5
0 8 6
-----
1 2 3
0 4 5
7 8 6
-----
1 2 3
4 0 5
7 8 6
-----
1 2 3
4 5 0
7 8 6
```



```
1 2 3
4 5 6
7 8 0
-----
Solved in 20 moves.
```

Conclusion: We have Successfully Executed the Program for 8-Puzzle problem using Python.

Viva Questions and Answers:

1. **What is the 8-puzzle problem?**
   Ans: It's a sliding puzzle consisting of a 3x3 grid with 8 numbered tiles and one blank space. The objective is to reach a goal state by sliding tiles.

2. **Which algorithms can be used to solve the 8-puzzle?**
   Ans: Breadth-First Search (BFS)
   Depth-First Search (DFS)
   A* Search (commonly used with heuristics)

3. **Why is A better than BFS or DFS for this problem?***
   Ans: A* uses heuristics to guide the search, making it faster and more efficient for large state spaces..

PRACTICAL NO: 5

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

Theory:

The **Water-Jug Problem** is a classic problem in Artificial Intelligence and problem-solving. It involves using two jugs with different capacities to measure out a specific amount of water using only the jugs and unlimited water supply.

For example:

- Jug A has capacity 4 liters
- Jug B has capacity 3 liters
- Goal: Measure **2 liters** of water.

Approach Used:
To solve the problem, we can use Breadth-First Search (BFS) or Depth-First Search (DFS) to explore all possible states until the goal is found.
The idea is to model each state as a node in a graph and generate its successors using valid operations.

Algorithm:

1. Start
   Initialize the jug capacities:
   Let jug1_capacity, jug2_capacity, and target be the inputs.

2. Define the State
   Represent the state as a pair (x, y) where:
   - x = current amount in Jug 1
   - y = current amount in Jug 2

3. Initialize
   - Create a queue for BFS and add the initial state (0, 0)
   - Create a visited set to track visited states

   - Create a dictionary to store the parent of each state (for tracing the solution path)

4. BFS Loop
   Repeat until the queue is empty:

   - Dequeue the front state (x, y)

   - If x == target or y == target:
     → Goal is reached, break

   - For each of the following valid operations, generate a new state:

     - Fill Jug 1: (jug1_capacity, y)

     - Fill Jug 2: (x, jug2_capacity)

     - Empty Jug 1: (0, y)

     - Empty Jug 2: (x, 0)

     - Pour Jug 1 → Jug 2

     - Pour Jug 2 → Jug 1

   - If a new state has not been visited:

     - Add it to the queue and visited set

     - Record its parent for path tracing

5. Trace Path (Optional)

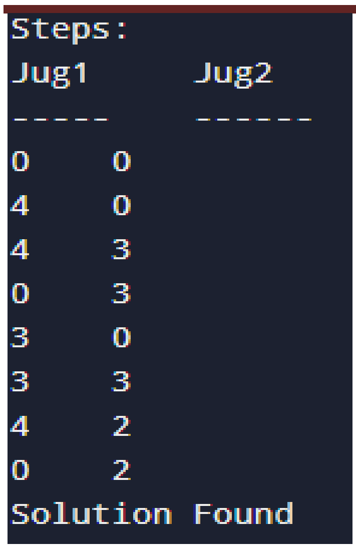
   - Use the parent dictionary to backtrack from the goal to the initial state and get the sequence of steps.

6. End

Program:

```
# 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(vol2, (jug1 - vol1))) or  # Pour
water from jug2 to jug1
     waterJug(vol1 - min(vol1, (jug2 - vol2)), vol2 + min(vol1, (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
```

Conclusion: We have Successfully Executed the program to Implement Water-Jug problem using Python.

Viva Questions and Answers:

1. **What is the Water-Jug problem?**
   Ans:The Water-Jug problem is a classic problem in artificial intelligence and algorithm design. It involves two jugs of different capacities and the goal is to measure out a specific amount of water using only these jugs, with unlimited water supply.

2. **Why is BFS preferred over DFS in this problem?**
   Ans:BFS is preferred because it finds the shortest sequence of steps to reach the goal. DFS may go deep into longer or even infinite paths before finding a solution, which is inefficient.
3. **Can the Water-Jug problem be unsolvable?**
   Ans: Yes. The problem is solvable only if the target volume is a multiple of the GCD (Greatest Common Divisor) of the two jug capacities and does not exceed the capacity of the larger jug.

4. **Can we delete a node without traversing the list?**
   Ans: No, unless we are given a direct pointer to the node and not just the value. In general, we must traverse the list to locate the node to delete it.

PRACTICAL NO: 6

Aim: Write a Program to Implement Travelling Salesman Problem using Python.

Theory:

The Travelling Salesman Problem (TSP) is one of the most famous problems in computer science and operations research. It focuses on optimization and is classified as an NP-Hard problem.
A salesman needs to visit a set of cities exactly once and return to the starting city. The goal is to find the shortest possible route that visits each city once and returns to the original city.

Problem Definition:
Given:
- A set of cities.
- The distance (or cost) between each pair of cities.

Objective:
- Find the shortest path that visits every city exactly once and returns to the starting city.

Applications:
- Logistics and delivery routing.
- Microchip manufacturing.
- DNA sequencing.
- Robotics path planning.

Limitations:
- Not suitable for very large number of cities (e.g., >20) due to exponential time.
- Approximation or heuristic methods are preferred for large-scale TSP.

Algorithm:

Input:
- A cost matrix cost[n][n] representing distances between n cities.

Output:
- The minimum cost of the tour that visits every city exactly once and returns to the starting city.

Steps:
1. Initialize Constants:
    - Let n be the number of cities.

- Let dp[2^n][n] be a 2D array to store minimum costs.
- dp[mask][i] means minimum cost to reach city i having visited cities represented by mask.

2. Base Case Initialization:
   - Set all values of dp to infinity (inf).
   - Set dp[1 << i][i] = cost[0][i] for all cities i ≠ 0.
3. Dynamic Programming Recurrence:
   - For all mask from 0 to 2^n - 1:
     - For all cities u (current city) in mask:
       - For all cities v not in mask:
         - Update:
           dp[mask | (1 << v)][v] = min(dp[mask | (1 << v)][v], dp[mask][u] + cost[u][v])
4. Compute Final Answer:
   - After filling the table, the answer is:

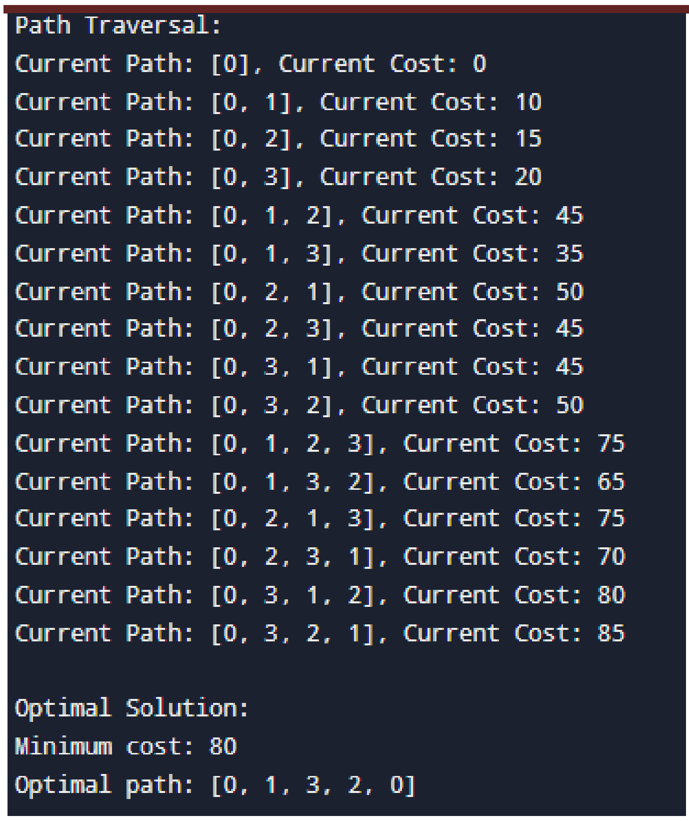
css
CopyEdit
min(dp[(1<<n)-1][i] + cost[i][0]) for all i from 1 to n-1
(This completes the tour by returning to the starting city 0)

5. Return the Minimum Cost.

Program:

```
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_city]
                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}")
```

Output:


```
Path Traversal:
Current Path: [0], Current Cost: 0
Current Path: [0, 1], Current Cost: 10
Current Path: [0, 2], Current Cost: 15
Current Path: [0, 3], Current Cost: 20
Current Path: [0, 1, 2], Current Cost: 45
Current Path: [0, 1, 3], Current Cost: 35
Current Path: [0, 2, 1], Current Cost: 50
Current Path: [0, 2, 3], Current Cost: 45
Current Path: [0, 3, 1], Current Cost: 45
Current Path: [0, 3, 2], Current Cost: 50
Current Path: [0, 1, 2, 3], Current Cost: 75
Current Path: [0, 1, 3, 2], Current Cost: 65
Current Path: [0, 2, 1, 3], Current Cost: 75
Current Path: [0, 2, 3, 1], Current Cost: 70
Current Path: [0, 3, 1, 2], Current Cost: 80
Current Path: [0, 3, 2, 1], Current Cost: 85

Optimal Solution:
Minimum cost: 80
Optimal path: [0, 1, 3, 2, 0]
```

Conclusion: We have Successfully Executed the program to Implement Travelling Salesman Problem using Python.

Viva Questions and Answers:

1. **What is the Travelling Salesman Problem (TSP)?**
   Ans: TSP is a classic optimization problem where a salesman must visit each city exactly once and return to the starting city with the shortest possible route.

2. **Is TSP a decision problem or an optimization problem?**
   Ans: Utilizes all available space in the array.Prevents the "false overflow" issue of linear queue.Efficient in fixed-size memory environments (e.g., embedded systems).

3. **What is the input to a TSP program?**
   Ans: A distance matrix or graph where nodes represent cities and edges represent the distance or cost between cities.

PRACTICAL NO: 7

Aim: Write a Program to Implement Tower of Hanoi using Python.

Theory:

The **Tower of Hanoi** is a classic **recursive problem** in computer science and mathematics. It involves moving a stack of disks from one rod (source) to another rod (destination) using an auxiliary rod, following specific rules.
Need to Convert Infix to Postfix
Infix expressions are natural for humans to read but difficult for computers to evaluate due to parentheses and operator precedence.
Postfix expressions:
- Do **not** need parentheses
- Are **easier and faster** to evaluate using a stack
- Obey **strict left-to-right evaluation**

Rules of the Tower of Hanoi:
1. Only **one disk** can be moved at a time.
2. A move consists of taking the **top disk** from one rod and placing it on **top of another rod**.
3. **No disk** may be placed on **top of a smaller disk**.

Applications:
- Understanding recursion and divide-and-conquer algorithms.
- Used in algorithm design, mathematical puzzles, and computer science education.

Algorithm:

Input:
- n: Number of disks
- source: The peg from which to move the disks (e.g., 'A')
- auxiliary: The peg to use for temporary storage (e.g., 'B')
- destination: The peg to move the disks to (e.g., 'C')

Steps:
1. **Start**
2. If n == 1 (base case):
   - Move disk from source to destination
3. Else:
   - Recursively move n-1 disks from source to auxiliary using destination as

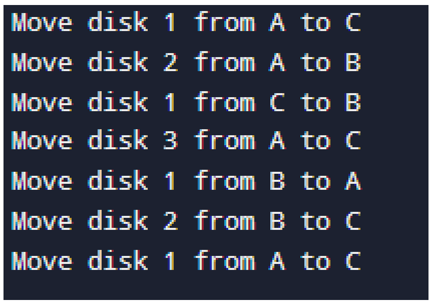
temporary
- Move the remaining 1 disk from source to destination
- Recursively move n-1 disks from auxiliary to destination using source as temporary

4. **End**

Program:

```
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
```

Conclusion: We have Successfully Executed Program to Implement Tower of Hanoi using Python.

Viva Questions and Answers:

1. **What is the Tower of Hanoi problem?**

Ans: The Tower of Hanoi is a classic recursive problem involving three pegs (A, B, C) and a number of disks of different sizes. The goal is to move all disks from the source peg (A) to the destination peg (C) using the auxiliary peg (B), following these rules:

2. Only one disk can be moved at a time.
3. A larger disk cannot be placed on a smaller disk.
4. Only the top disk of a peg can be moved.

2. **What is the base case in the recursive solution of Tower of Hanoi?**
   Ans**:** The base case is when there is only **one disk**. In that case, simply move it from the source to the destination peg.

3. **Can Tower of Hanoi be solved iteratively?**
   Ans: Yes, it can be solved iteratively using a **stack** or by following a specific sequence of moves, but it is typically solved recursively due to its natural recursive structure.

4. **How many total moves are required to solve Tower of Hanoi for n disks?**
   Ans: The total number of moves required is $\mathbf{2^n - 1}$.

PRACTICAL NO: 8

Aim: Write a Program to Implement Monkey Banana Problem using Python.
Theory:

The **Monkey and Banana** problem is a classic example of a problem-solving scenario in Artificial Intelligence (AI), especially in the domain of planning and reasoning. It was originally introduced to study goal-based agents and AI planning in a structured environment. In this problem, a monkey is placed in a room where bananas are hanging from the ceiling, and the monkey must find a sequence of actions to get the bananas using a box placed in the room. The challenge lies in finding the correct sequence of actions.

Problem Description

- The monkey is in a room.
- Bananas are hanging from the ceiling — the monkey cannot reach them from the ground.
- A box is present in the room — the monkey can climb on it to reach the bananas.
- The monkey can perform several actions:
    - Walk to the box.
    - Push the box to a location under the bananas.
    - Climb onto the box.
    - Grab the bananas.

Algorithm:

1. **Define the State Structure**:
    - A state can be represented as a tuple:

(Monkey_Location, Box_Location, Monkey_Position, Has_Banana)
Example: ("door", "middle", "floor", False)

2. **Initialize Initial State**:
    - Monkey_Location = "door"
    - Box_Location = "middle"
    - Monkey_Position = "floor"
    - Has_Banana = False
3. **Define Goal State**:
    - The monkey has the banana:

Has_Banana = True

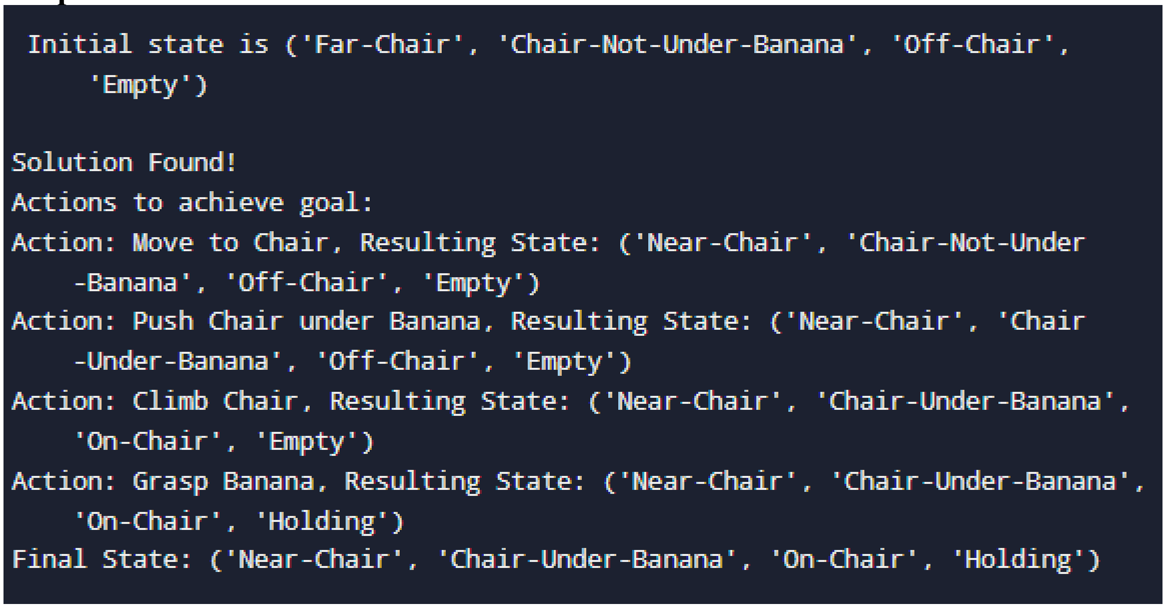
4. **Define Possible Actions with Preconditions and Effects**:
    - **Move(location)**: Monkey moves to a new location.
    - **Push(box_location, new_location)**: Monkey pushes the box to new location.
    - **ClimbUp()**: Monkey climbs onto the box.
    - **ClimbDown()**: Monkey climbs down from the box.

   - **Grasp()**: Monkey grabs the banana (only if on the box and under banana).
5. **Create a Queue or Stack for Search**:
   - Use BFS or DFS.
   - Store pairs of (state, action_path).
6. **While the queue is not empty**:
   - Remove the front node from the queue.
   - If the current state satisfies the goal condition:
     - Return the list of actions to reach this state.
   - Otherwise:
     - For each valid action from this state:
       - Generate a new state.
       - If the state is not visited, add it to the queue with the updated path.
7. **If queue is empty and goal not reached**:
   - Return "No solution found."

Program:

```
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 (current_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)
          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')
```

Conclusion: We have Successfully Executed the Program to Implement Monkey Banana Problem using Python

Viva Questions and Answers:

1. **What is the Monkey-Banana problem?**
   Ans: The Monkey-Banana problem is a classic problem in Artificial Intelligence where a monkey must retrieve bananas hanging from the ceiling using a box. It demonstrates goal-based agent planning by finding a sequence of actions (like move, push, climb, and grasp) to achieve the goal of getting the bananas.

2. **List the possible actions the monkey can perform.**
   Ans: The monkey can:

1. **Move(x)** – move to a location
2. **Push(x, y)** – push the box from x to y
3. **ClimbUp** – climb onto the box
4. **ClimbDown** – climb down from the box
5. **Grasp** – grasp the banana if on the box and under bananas

3. **What search strategy is suitable for this problem?**
   Ans: Both **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** can be used, but BFS is preferred if we want the shortest sequence of actions.

PRACTICAL NO: 9

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

Theory:
Alpha-Beta Pruning is an optimization technique for the **Minimax algorithm**, widely used in **Artificial Intelligence (AI)**, especially in **two-player turn-based games** like Tic-Tac-Toe, Chess, or Checkers. It reduces the number of nodes evaluated in the search tree, making decision-making more efficient without affecting the final result.

Purpose of Alpha-Beta Pruning

- To optimize the **Minimax algorithm**.
- To avoid unnecessary computations by **pruning** branches that do not influence the final decision.
- To improve **search efficiency**, especially in games with large decision trees.

Working Principle
Alpha-Beta pruning maintains two values during the search:

- **Alpha (α):** The best value that the maximizer currently can guarantee.

- **Beta (β):** The best value that the minimizer currently can guarantee.

Key rule:

- If **Beta ≤ Alpha**, further exploration of that node is **pruned** (cut off), because it cannot affect the final decision.

Algorithm:

Step 1: Start

Step 2: Define the Alpha-Beta function with the following parameters:

- node: current node in the tree
- depth: current depth of the node
- isMaximizingPlayer: Boolean, true for maximizer's turn, false for minimizer's
- alpha: best value that the maximizer can guarantee
- beta: best value that the minimizer can guarantee

Step 3: Check if terminal condition is met

(Depth is 0 or node is a terminal node)

- Return the heuristic value of the node

Step 4: If isMaximizingPlayer is True:

- Initialize maxEval = -∞
- For each child of the node:
  - Call the Alpha-Beta function recursively with:
    - depth - 1, isMaximizingPlayer = False, updated alpha, beta
  - Update maxEval = max(maxEval, eval)
  - Update alpha = max(alpha, eval)
  - If beta ≤ alpha: **Break** (prune branch)
- Return maxEval

Step 5: Else (Minimizing Player's Turn):

- Initialize minEval = +∞
- For each child of the node:
  - Call the Alpha-Beta function recursively with:
    - depth - 1, isMaximizingPlayer = True, updated alpha, beta
  - Update minEval = min(minEval, eval)
  - Update beta = min(beta, eval)
  - If beta ≤ alpha: **Break** (prune branch)
- Return minEval

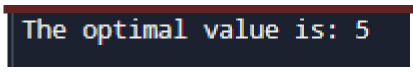
Step 6: End

Program:

```
"""
        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 maximizing, 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, values, 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 usage
if __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
```

Conclusion: We have Successfully Executed a Program to Implement Alpha-Beta Pruning using Python.

Viva Questions and Answers:

1. **What is Alpha-Beta Pruning?**
   Ans:Alpha-Beta Pruning is an optimization technique for the Minimax algorithm that eliminates branches in the game tree that don't affect the final decision, thereby reducing computation time.

2. **What are Alpha and Beta in Alpha-Beta Pruning?**
   Ans: **Alpha (α)**: The best (maximum) value that the maximizer can guarantee so far.
   Beta (β): The best (minimum) value that the minimizer can guarantee so far.

3. **How does Alpha-Beta Pruning improve Minimax performance?**
   Ans: It avoids evaluating parts of the tree that won't influence the final decision, thus reducing the number of nodes explored and improving efficiency.

4. **Can Alpha-Beta Pruning be applied to all types of games?**
   Ans: It is applicable to two-player, zero-sum, turn-based deterministic games like Chess, Checkers, and Tic-Tac-Toe.

PRACTICAL NO: 10

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

Theory:
The **8-Queens Problem** is a classic example of the **backtracking algorithm** in Artificial Intelligence and Computer Science. The problem is to place **eight queens** on an **8×8 chessboard** such that no two queens threaten each other. This means:

- No two queens can be in the same **row**,
- No two queens can be in the same **column**,
- No two queens can be on the same **diagonal**.

Problem Solving Approach
To solve this problem, we use **recursive backtracking**:

1. **Start from the first row** and try placing a queen in each column one by one.
2. Before placing a queen, **check if the position is safe** (i.e., it is not under attack from any previously placed queens).
3. If a safe position is found, **place the queen and move to the next row**.

4. If placing the queen leads to a solution, continue. If not, **backtrack** and try another column in the previous row.
5. Continue this process until all 8 queens are placed successfully or all possibilities are exhausted.

Applications

- This problem is a typical use-case of **constraint satisfaction problems**.
- It demonstrates **backtracking**, a foundational technique in AI.
- Solving N-Queens helps understand **optimization**, **search techniques**, and **state space exploration**.

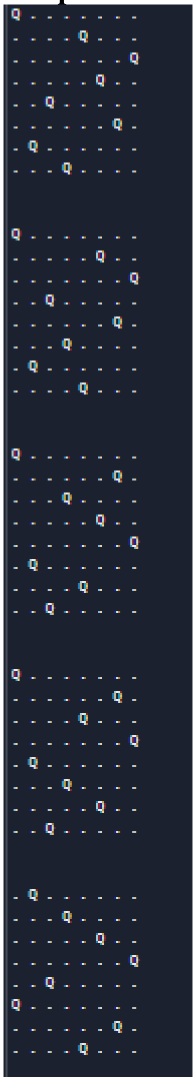
Algorithm:

```
1. Start
   Create an 8×8 chessboard initialized with empty values (0 or .).
2. Define a function is_safe(board, row, col)
      o Check if there is a queen in the same column in previous rows.
      o Check the upper-left diagonal for a queen.
      o Check the upper-right diagonal for a queen.
      o Return True if it is safe, otherwise False.
3. Define a recursive function solve(board, row)
      o If row == 8, all queens are placed → Print the board and return True.
      o For each column col from 0 to 7:
           ▪ If is_safe(board, row, col):
                ▪ Place a queen at board[row][col].
                ▪ Recursively call solve(board, row + 1).
                ▪ If recursive call returns True, return True.
                ▪ Else, Backtrack → Remove the queen from
                  board[row][col].
4. In the main function:
      o Initialize the board with 0 or ..
      o Call solve(board, 0) to begin placing from the first row.
      o If no solution is found, print "No solution exists".
5. End
6. End
```

Program:

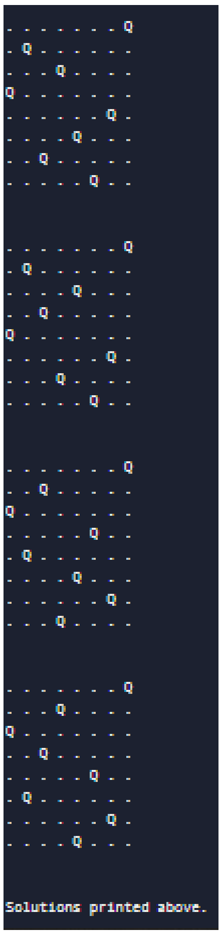
```
def printSolution(board):
    """Print the chessboard configuration."""
    for row in board:
        print(" ".join("Q" if col else "." for col in row))
    print("\n")

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
            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 . . . . .

Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .

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

……….......................................



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

. . . . . . . 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.
```

Conclusion: We have Successfully executed a Program to Implement 8-Queens Problem using Python.

Viva Questions and Answers:

1. **What is the 8-Queens Problem?**
   Ans: The 8-Queens Problem is a classic AI and backtracking problem where the goal is to place 8 queens on a standard 8×8 chessboard such that no two queens threaten each other. This means no two queens share the same row, column, or diagonal.

2. **Which algorithm is used to solve the 8-Queens Problem?**
   Ans: The 8-Queens Problem is solved using the **backtracking algorithm**, a form of depth-first search that tries possible solutions and backtracks when a constraint is violated.

3. **Why can't two queens be placed on the same diagonal?**
   Ans: In chess, a queen can move diagonally in all directions. So, placing two queens on the same diagonal means one can attack the other, violating the problem's constraint.

4. **What is backtracking?**
   Ans: Backtracking is an algorithmic technique for solving recursive problems by trying to build a solution incrementally and removing those solutions that fail to meet the constraints (i.e., backtrack and try a different option).