Artificial Intelligence Using Python [Lab Programs]

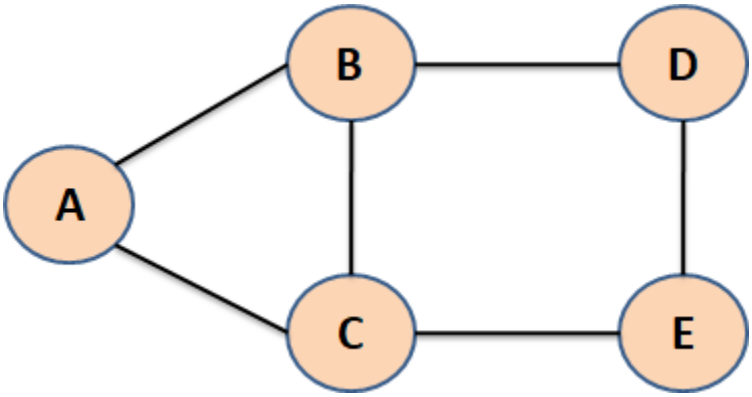
Exp1: Breadth First Search (BFS)

Aim: To write a Program to implement Breadth First Search (BFS) using Python.

Algorithm:

1. **Initialize a queue and enqueue the starting node.**
2. **Mark the starting node as visited.**
3. **While the queue is not empty:**
    - **Dequeue a node from the front.**
    - **Process the node (e.g., print it).**
    - **Enqueue all its adjacent unvisited nodes and mark them as visited.**
4. **Repeat until all reachable nodes have been visited.**

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)
        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

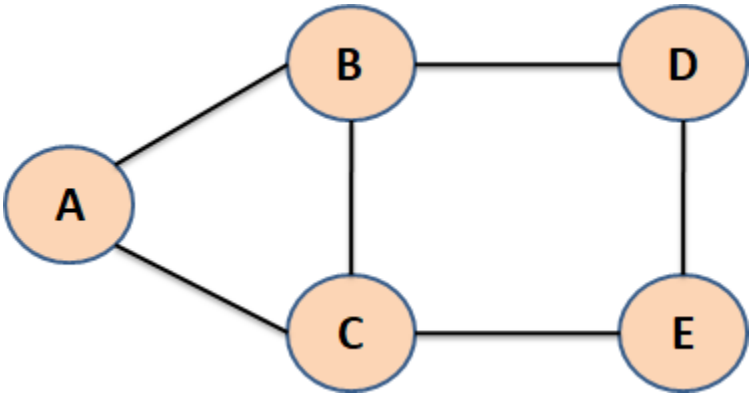
RESULT :

B A C D E

Exp2: Depth First Search (DFS)

AIM: To write a Program to implement Depth First Search (DFS) using Python.

Input Graph



Algorithm for DFS

1. Initialize an empty set visited and a stack.
2. Push the starting node to the stack.
3. While the stack is not empty:
    - Pop a node from the stack.
    - If the node is not visited:
        - Mark it as visited.
        - Process the node.
        - Push all unvisited neighbors onto the stack.
4. Repeat until all nodes are visited.

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
```

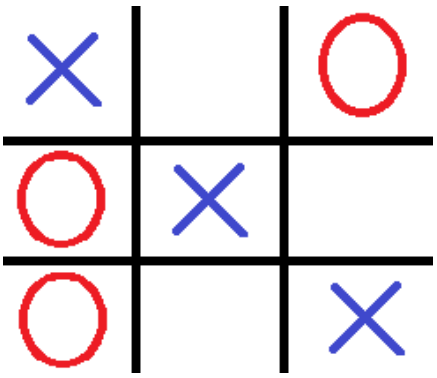
Sample Output 2:

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

Exp 3: Tic-Tac-Toe game

AIM: To write a Program to implement the Tic-Tac-Toe game using Python.

Input Graph :



Algorithm

Step 1: Initialize the Board

- Create a 3x3 grid (list of lists) initialized with empty spaces (`' '`).

Step 2: Define Functions

1. **Display the board:**
   - Print the current state of the board.
2. **Check for a win:**
   - Check all rows, columns, and diagonals to see if either player has won.
3. **Check for a draw:**
   - If all cells are filled and there's no winner, declare a draw.
4. **Take player input:**
   - Ask the user for a row and column number to place their mark (`X` or `O`).
   - Ensure the move is valid (cell is not already occupied).
5. **Switch turns:**
   - Alternate between `X` and `O` after each turn.

Step 3: Main Game Loop

- Repeat until there is a winner or a draw.
- Display the board after each move.
- Declare the winner or a draw when the game ends.

SOURCE CODE :

```
# 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):
    # display 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! ***
```

Exp 4: 8-Puzzle problem

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

Algorithm:

Step 1: Define the Problem State

- Represent the puzzle as a **3x3 matrix (list of lists)**.
- Define the **goal state** `[[1, 3, 0], [6, 8, 4], [7, 5, 2]]` where `0` represents the blank space).
- Track the **possible moves** (left, right, up, down).

Step 2: Implement a Search Algorithm

- Use *A search algorithm (A-star)** with the **Manhattan distance heuristic**:
    - **f(n) = g(n) + h(n)**, where:
        - `g(n)`: Number of moves taken from the initial state.
        - `h(n)`: Estimated cost to reach the goal (Manhattan distance).

Step 3: Define Helper Functions

1. **Find the blank space (`0`)**.
2. **Generate valid moves** and new states.
3. **Check if a state is the goal state**.
4. **Implement the heuristic function** (Manhattan distance).
5. **Use a priority queue (min-heap) to explore states efficiently**.

Step 4: Solve the Puzzle

- Start with the **initial state**.
- Use a **priority queue** (min-heap) to expand the state with the lowest `f(n)`.
- If the goal state is reached, return the sequence of moves.
- Continue until a solution is found or no solution exists.

SOURCE CODE :

```
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 2 3

4 5 0

7 8 6

- - - .

1 2 3

4 5 6

7 8 0

- - - .

Solved in 20 moves.

Exp 5: Water-Jug problem

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

Water Jug Problem

The Water Jug Problem is a classic AI problem where you are given two jugs of different capacities and need to measure a specific amount of water using them.

Problem Statement

- You have two jugs with capacities X liters and Y liters.
- Your goal is to measure exactly Z liters using the jugs.
- Allowed Operations:
    - Fill a jug.
    - Empty a jug.
    - Pour water from one jug to another until one is full or the other is empty.

Algorithm:

1. Create a queue to store the state of jugs (x, y).
2. Create a set to store visited states to avoid repetition.
3. Enqueue the initial state (0, 0) and mark it as visited.
4. While the queue is not empty:
    - Dequeue the current state (a, b).
    - If a == Z or b == Z, print the steps and return success.
    - Generate the next possible states by:
        - Filling jug X
        - Filling jug Y
        - Emptying jug X
        - Emptying jug Y
        - Pouring from X to Y
        - Pouring from Y to X
    - If the new state is not visited, add it to the queue and mark it as visited.
5. If no solution is found, return failure.

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 possi
ble 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, (ju
g1 - vol1))) or  # Pour water from jug2 to jug1
        waterJug(vol1 - min(vol1, (jug2 - vol2)), vol2 + min(vol1, (ju
g2 - vol2)))  # Pour water from jug1 to jug2
    )

print("Steps: ")
print("Jug1 \t Jug2 ")
print("----- \t ------")
waterJug(0, 0)
```

OUTPUT:

```
C:\Users\rites\PycharmProjects\PythonProject\.venv\Scripts\python.exe C:\Users\rites\AppData\Roami
Steps:
Jug1     Jug2
-----    ------
0    0
4    0
4    3
0    3
3    0
3    3
4    2
0    2
Solution Found

Process finished with exit code 0
```

Exp 6: Travelling Salesman Problem

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

Traveling Salesman Problem (TSP)

The **Travelling Salesman Problem (TSP)** is a classic optimization problem where a salesman is given a list of cities and must find the shortest possible route that visits each city exactly once and returns to the starting city.

Problem Statement

- Input:
  - A set of `N` cities.
  - A distance matrix where `dist[i][j]` represents the distance between cities i and `j`.
- Goal:
  - Find the shortest possible route that visits each city once and returns to the starting point.

Algorithm (Using Dynamic Programming + Bitmasking)

1. Define a 2D array `dp[mask][i]` where:
   - `mask` = Binary representation of visited cities.
   - `i` = Current city.
2. Initialize `dp[mask][i]` to `infinity` except `dp[1][0] = 0` (starting from city 0).
3. For each state (`mask`):
   - Try to extend the tour by moving to an unvisited city `j`.
   - Update `dp[new_mask][j]` using the recurrence:

     dp[new_mask][j]=minf_0(dp[new_mask][j],dp[mask][i]+dist[i][j])dp[new_mask][j] = \min(dp[new_mask][j], dp[mask][i] + dist[i][j])dp[new_mask][j]=min(dp[new_mask][j],dp[mask][i]+dist[i][j])
4. Final answer = Minimum distance to visit all cities and return to starting point:

   minf_0(dp[2N−1][i]+dist[i][0])\min(dp[2^N - 1][i] + dist[i][0])min(dp[2N−1][i]+dist[i][0])

SOURCE CODE :

```python
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:

C:\Users\rites\PycharmProjects\PythonProject\.venv\Scripts\python.exe
C:\Users\rites\AppData\Roaming\JetBrains\PyCharmCE2024.3\scratches\scratch.py
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]

Process finished with exit code 0

Exp 7: Travelling Salesman Problem

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

Tower of Hanoi Problem

The **Tower of Hanoi** is a classic recursive problem where the goal is to move a set of disks from one peg to another using an auxiliary peg, following specific rules:

Problem Statement

- There are **three pegs** (A, B, C) and **n disks** of different sizes stacked on peg A.
- Goal: Move all disks from peg A to peg C using peg B as an auxiliary.
- Constraints:
    1. Only one disk can be moved at a time.
    2. A disk can only be placed on top of a larger disk.
    3. A disk can only be moved from the top of a peg.

Algorithm

1. Base Case:
    - If there's only one disk, move it directly from the source peg to the destination peg.
2. Recursive Case:
    - Step 1: Move `n - 1` disks from the source peg to the auxiliary peg using the destination peg.
    - Step 2: Move the largest disk from the source peg to the destination peg.
    - Step 3: Move the `n - 1` disks from the auxiliary peg to the destination peg using the source peg.

SOURCE CODE :

```
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:

```
C:\Users\rites\PycharmProjects\PythonProject\.venv\Scripts\python.exe C:\Users\rites\AppData\Roamir
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

Process finished with exit code 0
```

Exp 8: Monkey Banana Problem

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

Monkey Banana Problem

The **Monkey Banana Problem** is a classic AI problem where a monkey in a room wants to get a bunch of bananas hanging from the ceiling. The monkey needs to use a chair to reach the bananas by performing a sequence of actions.

Problem Statement

1. A monkey is in a room.
2. A chair is present in the room.
3. Bananas are hanging from the ceiling, out of the monkey's reach.
4. The monkey needs to:
    - Move to the chair.
    - Push the chair under the bananas.
    - Climb onto the chair.
    - Grab the bananas.

States and Actions

1. **Initial State** – Monkey is at a certain location, bananas are hanging, chair is at another location.
2. **Goal State** – Monkey is holding the bananas.
3. **Possible Actions**:
    - Walk to the chair.
    - Push the chair to the bananas.
    - Climb onto the chair.
    - Grab the bananas.

Algorithm (Using State Space Search)

1. Define the possible states:
    - Location of the monkey.
    - Location of the chair.
    - Whether the monkey is on the chair.
    - Whether the monkey is holding the bananas.
2. Define possible actions:
    - Move
    - Push
    - Climb

- Grab

3. Use **Breadth-First Search (BFS)** to explore all possible states.
4. If the state where the monkey holds the bananas is reached → Success!
5. If no state leads to success → Fail.

SOURCE CODE :

```
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', 'Hol
ding')     # 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 s
tate[1] != 'Chair-Under-Banana' else None,
        "Climb Chair": lambda state: ('Near-Chair', 'Chair-Under-Banan
a', '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-Bana
na', 'On-Chair',  'Holding') if state[0] == 'Near-Chair' and state[1]
== 'Chair-Under-Banana' and state[2] == 'On-Chair' and state[3] !='Hol
ding' else None
    }

    # BFS to explore states
    from collections import deque
    dq = deque([(initial_state, [])])  # Each element is (current_stat
e, 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: {act
ion_name}, Resulting State: {next_state}"]))

    print("No solution found.")

# Run the program
monkey_banana_problem()
```

OUTPUT:

Exp 9: Alpha-Beta Pruning

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

Alpha-Beta Pruning

Alpha-Beta Pruning is an optimization technique for the **Minimax algorithm**. It reduces the number of nodes evaluated by the Minimax algorithm in a search tree, improving efficiency without affecting the final result.

Problem Statement

1. Given a two-player game tree (e.g., Tic-Tac-Toe, Chess), the goal is to:
    - Find the best possible move for the maximizing player while assuming the opponent plays optimally.
2. Alpha-beta pruning improves Minimax by "pruning" branches that cannot affect the final decision.

Key Terms

- **Alpha** – Best value the maximizer can guarantee so far (initialized to −∞).
- **Beta** – Best value the minimizer can guarantee so far (initialized to +∞).
- **Pruning** – Stopping evaluation of a branch when the outcome is already determined.

Algorithm

1. Start with initial alpha = −∞ and beta = +∞.
2. Apply the Minimax algorithm with the following pruning rules:
    - If the current value exceeds beta → **Stop searching (prune the branch)**.
    - If the current value is less than alpha → **Stop searching (prune the branch)**.
3. Update alpha and beta values during the search:
    - If the value is greater than alpha → Update alpha.
    - If the value is less than beta → Update beta.
4. Continue until all possible moves are evaluated or pruned.
5. Return the optimal value for the maximizing player.

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 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, a
lpha, 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, F
alse, 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, T
rue, 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.i
nf, math.inf)
    print(f"The optimal value is: {optimal_value}")
```

OUTPUT:

```
C:\Users\rites\PycharmProjects\PythonProject\.venv\Scripts\python.exe
The optimal value is: 5


Process finished with exit code 0
```

Exp 10: 8-Queens Problem

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

8-Queens Problem

The **8-Queens Problem** is a classic combinatorial problem where the goal is to place 8 queens on a **chessboard** such that:

✅ No two queens threaten each other.

✅ No two queens share the same row, column, or diagonal.

Problem Statement

- Input: An 8 × 8 chessboard.
- Goal: Place 8 queens on the board such that:
  - No two queens are in the same row.
  - No two queens are in the same column.
  - No two queens are on the same diagonal.

Algorithm (Backtracking)

1. Start with an empty chessboard.
2. Place a queen in the first row.
3. Try placing a queen in the next row such that it is not attacked by any other queen.
4. If placing a queen is not possible:
   - Backtrack to the previous row.
   - Try a different column in that row.
5. Repeat steps until all 8 queens are placed.
6. If all queens are placed → Success!
7. If no valid solution exists → Failure.

SOURCE CODE :

```
N = 8
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 . . .


Solutions printed above.
```