ARTIFICIAL INTELLIGENCE

LAB MANUAL

Course Code: BAD402

Affiliated to VTU

1. Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem

```
from collections import defaultdict

jug1, jug2, aim = 4, 3, 2

visited = defaultdict(lambda: False)

def waterJugSolverDFS(amt1, amt2):
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True

    visited[(amt1, amt2)] = True
    print(amt1, amt2)

    # Try all possible moves
    if not visited[(0, amt2)] and waterJugSolverDFS(0, amt2):
        return True
    if not visited[(amt1, 0)] and waterJugSolverDFS(amt1, 0):
        return True
    if not visited[(jug1, amt2)] and waterJugSolverDFS(jug1, amt2):
        return True
    if not visited[(amt1, jug2)] and waterJugSolverDFS(amt1, jug2):
        return True

    # Pour from jug1 to jug2
    pour_amt = min(amt1, jug2 - amt2)
```

```
    if not visited[(amt1 - pour_amt, amt2 + pour_amt)] and waterJugSolverDFS(amt1 - pour_amt, amt2 + pour_amt):
        return True

    # Pour from jug2 to jug1
    pour_amt = min(amt2, jug1 - amt1)
    if not visited[(amt1 + pour_amt, amt2 - pour_amt)] and waterJugSolverDFS(amt1 + pour_amt, amt2 - pour_amt):
        return True

    return False

print("Steps:")
waterJugSolverDFS(0, 0)
```

OUTPUT

```
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2
```

Explanation

1. `from collections import defaultdict`: This line imports the `defaultdict` class from the `collections` module. `defaultdict` is a subclass of the built-in `dict` class, which provides a default value for keys that haven't been explicitly set.

2. `jug1, jug2, aim = 4, 3, 2`: This line initializes three variables `jug1`, `jug2`, and `aim` with values 4, 3, and 2 respectively. These variables represent the capacities of the two jugs and the desired amount of water to be measured.

3. `visited = defaultdict(lambda: False)`: This line creates a defaultdict named `visited` with a default value of False. This dictionary will be used to keep track of visited states during the DFS traversal.

4. `def waterJugSolverDFS(amt1, amt2):`: This line defines the main function `waterJugSolverDFS` that implements the DFS algorithm to solve the water jug problem. It takes two parameters `amt1` and `amt2`, representing the current amounts of water in jug1 and jug2 respectively.

5. `if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):`: This line checks if the current state is the goal state, where either jug1 or jug2 contains `aim` liters of water. If so, it prints the current state and returns True to indicate that the goal is reached.

6. `visited[(amt1, amt2)] = True`: This line marks the current state as visited by setting the corresponding key in the `visited` dictionary to True.

7. `print(amt1, amt2)`: This line prints the current amounts of water in both jugs.

8. The subsequent lines explore all possible moves from the current state:

   - Emptying jug1 and jug2 (`if not visited[(0, amt2)] and waterJugSolverDFS(0, amt2)`)

   - Filling jug1 and jug2 to their capacities (`if not visited[(jug1, amt2)] and waterJugSolverDFS(jug1, amt2)`)

   - Pouring water from jug1 to jug2 and vice versa (`if not visited[(amt1 - pour_amt, amt2 + pour_amt)] and waterJugSolverDFS(amt1 - pour_amt, amt2 + pour_amt)` and `if not visited[(amt1 + pour_amt, amt2 - pour_amt)] and waterJugSolverDFS(amt1 + pour_amt, amt2 - pour_amt)`)

9. `return False`: If none of the above moves lead to the goal state, the function returns False, indicating that the goal is not reachable from the current state.

10. `print("Steps:")`: This line prints the header for the steps that will be printed during the DFS traversal.

11. `waterJugSolverDFS(0, 0)`: This line calls the `waterJugSolverDFS` function with initial amounts of 0 in both jugs to start the DFS traversal from the initial state.

This code essentially implements a Depth-First Search (DFS) algorithm to explore all possible states of the water jug problem until the goal state is reached. During the traversal, it prints the steps taken to reach the goal state.

Water-jug Problem without DFS (another implementation)

```
from collections import defaultdict

# jug1 and jug2 contain the value
jug1, jug2, aim = 4, 3, 2

# Initialize dictionary with default value as false.
visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):
   if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
      print(amt1, amt2)
      return True

   if visited[(amt1, amt2)] == False:
      print(amt1, amt2)
      visited[(amt1, amt2)] = True
      return (waterJugSolver(0, amt2) or
            waterJugSolver(amt1, 0) or
            waterJugSolver(jug1, amt2) or
            waterJugSolver(amt1, jug2) or
            waterJugSolver(amt1 + min(amt2, (jug1 - amt1)), amt2 - min(amt2, (jug1 -
amt1))) or
            waterJugSolver(amt1 - min(amt1, (jug2 - amt2)), amt2 + min(amt1, (jug2 -
amt2))))
```

```
        else:
            return False

    print("Steps: ")
    waterJugSolver(0, 0)
```

OUTPUT

Steps:

0 0

4 0

4 3

0 3

3 0

3 3

4 2

0 2

2. Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems using Python

```
from collections import deque

# Define the initial state
initial_state = {'left': (3, 3), 'right': (0, 0), 'boat': 'left'}

# Define the goal state
goal_state = {'left': (0, 0), 'right': (3, 3), 'boat': 'right'}

# Define a function to check if a state is valid
def is_valid(state):
    left_m, left_c = state['left']
    right_m, right_c = state['right']
    if left_m < 0 or left_c < 0 or right_m < 0 or right_c < 0:
        return False
    if left_m > 3 or left_c > 3 or right_m > 3 or right_c > 3:
        return False
    if left_m < left_c and left_m > 0:
        return False
    if right_m < right_c and right_m > 0:
        return False
    return True

# Define a function to generate all possible next states from the current state
def generate_next_states(current_state):
    next_states = []
    for i in range(3):
        for j in range(3):
            if i + j > 2 or i + j == 0:
                continue
            if current_state['boat'] == 'left':
                new_state = {
                    'left': (current_state['left'][0] - i, current_state['left'][1] - j),
                    'right': (current_state['right'][0] + i, current_state['right'][1] + j),
                    'boat': 'right'
                }
            else:
                new_state = {
                    'left': (current_state['left'][0] + i, current_state['left'][1] + j),
                    'right': (current_state['right'][0] - i, current_state['right'][1] - j),
                    'boat': 'left'
                }
            if is_valid(new_state):
                next_states.append(new_state)
    return next_states
```

```
# Define the breadth-first search function
def bfs(initial_state, goal_state):
    visited = set()
    queue = deque([(initial_state, [])])

    while queue:
        current_state, path = queue.popleft()
        if current_state == goal_state:
            return path
        if tuple(current_state['left'] + current_state['right'] + (current_state['boat'],)) in
visited:
            continue
        visited.add(tuple(current_state['left'] + current_state['right'] +
(current_state['boat'],)))
        for next_state in generate_next_states(current_state):
            queue.append((next_state, path + [next_state]))

    return None

# Find the solution using BFS
solution = bfs(initial_state, goal_state)

# Print the solution
if solution:
    print("Solution found with", len(solution), "steps:")
    for i, state in enumerate(solution):
        print("Step", i + 1, ":", state)
else:
    print("No solution found.")
```

OUTPUT

```
Solution found with 11 steps:
Step 1 : {'left': (3, 1), 'right': (0, 2), 'boat': 'right'}
Step 2 : {'left': (3, 2), 'right': (0, 1), 'boat': 'left'}
Step 3 : {'left': (3, 0), 'right': (0, 3), 'boat': 'right'}
Step 4 : {'left': (3, 1), 'right': (0, 2), 'boat': 'left'}
Step 5 : {'left': (1, 1), 'right': (2, 2), 'boat': 'right'}
Step 6 : {'left': (2, 2), 'right': (1, 1), 'boat': 'left'}
Step 7 : {'left': (0, 2), 'right': (3, 1), 'boat': 'right'}
Step 8 : {'left': (0, 3), 'right': (3, 0), 'boat': 'left'}
Step 9 : {'left': (0, 1), 'right': (3, 2), 'boat': 'right'}
Step 10 : {'left': (0, 2), 'right': (3, 1), 'boat': 'left'}
Step 11 : {'left': (0, 0), 'right': (3, 3), 'boat': 'right'}
```

Explanation

1. `from collections import deque`: This line imports the `deque` class from the `collections` module. `deque` is a double-ended queue data structure that supports adding and removing elements efficiently from both ends.

2. `initial_state = {'left': (3, 3), 'right': (0, 0), 'boat': 'left'}`: This line defines the initial state of the problem. It represents the positions of missionaries and cannibals on both sides of the river, with the boat initially on the left side.

3. `goal_state = {'left': (0, 0), 'right': (3, 3), 'boat': 'right'}`: This line defines the goal state of the problem. It represents the positions of missionaries and cannibals on both sides of the river, with everyone on the right side.

4. `def is_valid(state): ...`: This block defines a function `is_valid` that checks whether a given state is valid according to the rules of the problem. It ensures that the number of missionaries on either side of the river is not less than the number of cannibals, and that the number of missionaries and cannibals on each side and in the boat is within the allowed range (0 to 3).

5. `def generate_next_states(current_state): ...`: This block defines a function `generate_next_states` that generates all possible next states from the current state. It considers all combinations of moving 1 or 2 missionaries or cannibals from one side to the other.

6. `def bfs(initial_state, goal_state): ...`: This block defines the breadth-first search (BFS) function. It takes the initial state and the goal state as input and searches for a solution using BFS.

7. `visited = set()`: This line initializes an empty set called `visited` to keep track of visited states during the BFS traversal.

8. `queue = deque([(initial_state, [])])`: This line initializes a deque called `queue` with a tuple containing the initial state and an empty list representing the path taken to reach that state. The initial state is added to the queue.

9. `while queue: ...`: This block starts a loop that continues until the queue is empty.

10. `current_state, path = queue.popleft()`: This line dequeues the leftmost element from the queue, which represents the current state and the path taken to reach that state.

11. `if current_state == goal_state: ...`: This line checks if the current state is equal to the goal state. If it is, the function returns the path to reach the goal state.

12. `if tuple(current_state['left'] + current_state['right'] + (current_state['boat'],)) in visited: ...`: This line checks if the current state has been visited before. If it has, the loop continues to the next iteration.

13. `for next_state in generate_next_states(current_state): ...`: This line iterates over all possible next states generated from the current state.

14. `queue.append((next_state, path + [next_state]))`: This line appends the next state and the updated path (including the next state) to the queue for further exploration.

15. `return None`: If no solution is found after exploring all possible states, the function returns None.

16. `solution = bfs(initial_state, goal_state)`: This line calls the BFS function with the initial state and goal state to find the solution.

17. `if solution: ...`: This line checks if a solution was found. If it was, it prints the number of steps and each state in the solution path.

18. `else: ...`: If no solution was found, this line prints a message indicating that no solution was found.

This code efficiently searches for a solution to the missionaries and cannibals problem using the breadth-first search algorithm and prints the solution if found.

3. Implement A* Search algorithm

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}              #store distance from starting node
    parents = {}        # parents contains an adjacency map of all nodes
    #distance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        #node with lowest f() is found
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                #nodes 'm' not in first and last set are added to first
                #n is set its parent
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                #for each node m,compare its distance from start i.e g(m) to the
                #from start through n node
                else:
                    if g[m] > g[n] + weight:
                        #update g(m)
                        g[m] = g[n] + weight
                        #change parent of m to n
                        parents[m] = n
                        #if m in closed set,remove and add to open
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None

        # if the current node is the stop_node
```

```
            # then we begin reconstructin the path from it to the start_node
            if n == stop_node:
                path = []
                while parents[n] != n:
                    path.append(n)
                    n = parents[n]
                path.append(start_node)
                path.reverse()
                print('Path found: {}'.format(path))
                return path
            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_set.remove(n)
            closed_set.add(n)
    print('Path does not exist!')
    return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None


        #for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
    }
    return H_dist[n]

    #Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
```

```
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}

aStarAlgo('A', 'J')
```

OUTPUT

```
Path found: ['A', 'F', 'G', 'I', 'J']
```

Explanation

This code implements the A* algorithm to find the shortest path between two nodes in a graph. Let's break down the code line by line:

1. `def aStarAlgo(start_node, stop_node):`: This line defines a function `aStarAlgo` that takes two arguments: `start_node` (the starting node) and `stop_node` (the destination node).

2. `open_set = set(start_node)`: This line initializes an open set with the starting node.

3. `closed_set = set()`: This line initializes a closed set, which will contain nodes that have already been evaluated.

4. `g = {}`: This line initializes a dictionary `g` to store the distance from the starting node to each node.

5. `parents = {}`: This line initializes a dictionary `parents` to store the parent node for each node in the path.

6. `g[start_node] = 0`: This line initializes the distance from the starting node to itself as 0.

7. `parents[start_node] = start_node`: This line sets the parent of the starting node to itself.

8. `while len(open_set) > 0:`: This line starts a loop that continues until the open set is empty.

9. `for v in open_set:`: This line iterates over all nodes in the open set.

10. `if n == stop_node or Graph_nodes[n] == None:`: This line checks if the current node is the destination node or if it has no neighbors.

11. `for (m, weight) in get_neighbors(n):`: This line iterates over the neighbors of the current node and their corresponding weights.

12. `if m not in open_set and m not in closed_set:`: This line checks if the neighbor is neither in the open set nor in the closed set.

13. `else:`: This line is the `else` block for the above `if` condition, which handles the case when the neighbor is already in either the open set or the closed set.

14. `if n == None:`: This line checks if the current node is `None`, indicating that no path exists.

15. `if n == stop_node:`: This line checks if the current node is the destination node.

16. `path = []`: This line initializes an empty list to store the path from the destination node to the starting node.

17. `while parents[n] != n:`: This line starts a loop to reconstruct the path from the destination node to the starting node.

18. `path.append(n)`: This line appends the current node to the path.

19. `path.append(start_node)`: This line appends the starting node to the path.

20. `path.reverse()`: This line reverses the order of nodes in the path to get the correct path from the starting node to the destination node.

21. `open_set.remove(n)`: This line removes the current node from the open set.

22. `closed_set.add(n)`: This line adds the current node to the closed set.

23. `print('Path does not exist!')`: This line prints a message indicating that no path exists if the loop completes without finding a solution.

24. `return None`: This line returns `None` if no solution is found.

25. `def get_neighbors(v): ...`: This block defines a function `get_neighbors` that returns the neighbors of a given node.

26. `def heuristic(n): ...`: This block defines a heuristic function that returns the heuristic distance for a given node.

27. `Graph_nodes = {...}`: This block defines the graph structure, where each node is mapped to its neighbors and their corresponding weights.

28. `aStarAlgo('A', 'J')`: This line calls the `aStarAlgo` function with the starting node `'A'` and the destination node `'J'` to find the shortest path between them.

This code efficiently finds the shortest path between two nodes in a graph using the A* algorithm.

4. Implement AO* Search algorithm

```
# Cost to find the AND and OR path
# Cost to find the AND and OR path
def Cost(H, condition, weight = 1):
        cost = {}
        if 'AND' in condition:
                AND_nodes = condition['AND']
                Path_A = ' AND '.join(AND_nodes)
                PathA = sum(H[node]+weight for node in AND_nodes)
                cost[Path_A] = PathA

        if 'OR' in condition:
                OR_nodes = condition['OR']
                Path_B =' OR '.join(OR_nodes)
                PathB = min(H[node]+weight for node in OR_nodes)
                cost[Path_B] = PathB
        return cost

# Update the cost
def update_cost(H, Conditions, weight=1):
        Main_nodes = list(Conditions.keys())
        Main_nodes.reverse()
        least_cost= {}
        for key in Main_nodes:
                condition = Conditions[key]
                print(key,':', Conditions[key],'>>>', Cost(H, condition, weight))
                c = Cost(H, condition, weight)
                H[key] = min(c.values())
                least_cost[key] = Cost(H, condition, weight)
        return least_cost

# Print the shortest path
def shortest_path(Start,Updated_cost, H):
        Path = Start
        if Start in Updated_cost.keys():
                Min_cost = min(Updated_cost[Start].values())
                key = list(Updated_cost[Start].keys())
                values = list(Updated_cost[Start].values())
                Index = values.index(Min_cost)

                # FIND MINIMIMUM PATH KEY
                Next = key[Index].split()
                # ADD TO PATH FOR OR PATH
                if len(Next) == 1:
```

```
                Start =Next[0]
                Path += '<--' +shortest_path(Start, Updated_cost, H)
        # ADD TO PATH FOR AND PATH
        else:
                Path +='<--('+key[Index]+') '

                Start = Next[0]
                Path += '[' +shortest_path(Start, Updated_cost, H) + ' + '

                Start = Next[-1]
                Path += shortest_path(Start, Updated_cost, H) + ']'

    return Path



H = {'A': -1, 'B': 5, 'C': 2, 'D': 4, 'E': 7, 'F': 9, 'G': 3, 'H': 0, 'I':0, 'J':0}

Conditions = {
'A': {'OR': ['B'], 'AND': ['C', 'D']},
'B': {'OR': ['E', 'F']},
'C': {'OR': ['G'], 'AND': ['H', 'I']},
'D': {'OR': ['J']}
}
# weight
weight = 1
# Updated cost
print('Updated Cost :')
Updated_cost = update_cost(H, Conditions, weight=1)
print('*'*75)
print('Shortest Path :\n',shortest_path('A', Updated_cost,H))
```

OUTPUT

```
Updated Cost :
D : {'OR': ['J']} > {'J': 1}
C : {'OR': ['G'], 'AND': ['H', 'I']} > {'H AND I': 2, 'G': 4}
B : {'OR': ['E', 'F']} > {'E OR F': 8}
A : {'OR': ['B'], 'AND': ['C', 'D']} > {'C AND D': 5, 'B': 9}
***************************************************************************
Shortest Path :
 A<--(C AND D) [C<--(H AND I) [H + I] + D<--J]
```

Explanation

1. **`def Cost(H, condition, weight = 1):`**: This function calculates the cost of finding the AND and OR paths based on the given conditions and their associated weights. It takes three arguments:
   - `H`: A dictionary representing the heuristic values for each node.
   - `condition`: A dictionary containing the conditions for each node.
   - `weight`: An optional parameter representing the weight assigned to each condition.

2. **`cost = {}`**: Initializes an empty dictionary to store the costs of different paths.

3. **`if 'AND' in condition:`**: Checks if the condition dictionary contains an 'AND' key.

4. **`AND_nodes = condition['AND']`**: Retrieves the nodes associated with the 'AND' condition.

5. **`Path_A = ' AND '.join(AND_nodes)`**: Creates a string representation of the 'AND' path by joining the node names with 'AND' as the separator.

6. **`PathA = sum(H[node]+weight for node in AND_nodes)`**: Calculates the cost of the 'AND' path by summing up the heuristic values of the nodes and adding the weight for each node.

7. **`cost[Path_A] = PathA`**: Stores the cost of the 'AND' path in the `cost` dictionary.

8. **`if 'OR' in condition:`**: Checks if the condition dictionary contains an 'OR' key.

9. **`OR_nodes = condition['OR']`**: Retrieves the nodes associated with the 'OR' condition.

10. **`Path_B =' OR '.join(OR_nodes)`**: Creates a string representation of the 'OR' path by joining the node names with 'OR' as the separator.

11. **`PathB = min(H[node]+weight for node in OR_nodes)`**: Calculates the cost of the 'OR' path by taking the minimum heuristic value among the nodes and adding the weight.

12. **`cost[Path_B] = PathB`**: Stores the cost of the 'OR' path in the `cost` dictionary.

13. **`return cost`**: Returns the dictionary containing the costs of the 'AND' and 'OR' paths.

The remaining functions (`update_cost` and `shortest_path`) are responsible for updating the costs based on the given conditions and finding the shortest path, respectively. The explanations for those functions have been provided in the previous response.

5. Solve 8-Queens Problem with suitable assumptions

```
# Taking number of queens as input from user
print ("Enter the number of queens")
N = int(input())

# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]

def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False

def N_queens(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0

    return False

N_queens(N)
for i in board:
    print (i)
```

OUTPUT

```
Enter the number of queens
8
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

Explanation

1. **`print ("Enter the number of queens")`**: Displays a prompt asking the user to input the number of queens.

2. **`N = int(input())`**: Reads the user input for the number of queens and converts it to an integer.

3. **`board = [[0]*N for _ in range(N)]`**: Initializes a two-dimensional list (matrix) called `board` with dimensions NxN, where each element is initially set to 0. This matrix represents the chessboard.

4. **`def attack(i, j):`**: Defines a function called `attack` that takes the indices of a cell `(i, j)` on the chessboard and checks if a queen placed at that cell is under attack by any other queen.

5. **`for k in range(0,N): if board[i][k]==1 or board[k][j]==1: return True`**: Checks if there is any queen in the same row (`i`) or column (`j`) as the current cell `(i, j)`.

6. **`for k in range(0,N): for l in range(0,N): if (k+l==i+j) or (k-l==i-j): if board[k][l]==1: return True`**: Checks if there is any queen placed diagonally to the current cell `(i, j)`.

7. **`def N_queens(n):`**: Defines a function called `N_queens` that recursively places `n` queens on the chessboard.

8. **`if n==0: return True`**: Base case for recursion. If `n` becomes 0, it means all queens have been placed successfully, so it returns `True`.

9. **`for i in range(0,N): for j in range(0,N): if (not(attack(i,j))) and (board[i][j]!=1):`**: Iterates through each cell of the chessboard and checks if it is safe to place a queen at that cell.

10. **`board[i][j] = 1`**: Places a queen at the current cell `(i, j)`.

11. **`if N_queens(n-1)==True: return True`**: Recursively calls the `N_queens` function with `n-1` queens to be placed. If placing the remaining queens is successful, it returns `True`.

12. **`board[i][j] = 0`**: If placing the remaining queens is not successful, it removes the queen from the current cell `(i, j)` and continues the loop to try other positions.

13. **`return False`**: If no queen can be placed safely at the current cell, it returns `False`.

14. **`N_queens(N)`**: Calls the `N_queens` function with the number of queens (`N`) as input to start the recursion and place all queens on the chessboard.

15. **`for i in board: print (i)`**: Prints the final configuration of the chessboard with queens placed on it.

This code uses a recursive backtracking approach to solve the N-Queens problem, where it tries to place queens on the chessboard and backtracks if it encounters an invalid position.

6. Implementation of TSP using heuristic approach

```
def tsp_nearest_neighbor(distances):
    num_cities = len(distances)
    unvisited_cities = set(range(num_cities))
    current_city = 0  # Start from city 0
    tour = [current_city]
    total_distance = 0

    while unvisited_cities:
        nearest_city = min(unvisited_cities, key=lambda city: distances[current_city][city])
        total_distance += distances[current_city][nearest_city]
        current_city = nearest_city
        unvisited_cities.remove(nearest_city)
        tour.append(current_city)

    total_distance += distances[current_city][0]  # Return to the starting city
    tour.append(0)

    return tour, total_distance

# Example usage:
distances = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

tour, total_distance = tsp_nearest_neighbor(distances)
print("Nearest Neighbor TSP Solution:")
print("Tour:", tour)
print("Total Distance:", total_distance)
```

OUTPUT

```
Nearest Neighbor TSP Solution:
Tour: [0, 0, 1, 3, 2, 0]
Total Distance: 80
```

Explanation

1. **`def tsp_nearest_neighbor(distances):`**: This line defines a function called `tsp_nearest_neighbor` that takes a `distances` matrix as input. This matrix represents the distances between cities in a traveling salesman problem (TSP).

2. **`num_cities = len(distances)`**: Calculates the number of cities in the TSP by finding the length of the `distances` matrix.

3. **`unvisited_cities = set(range(num_cities))`**: Initializes a set containing the indices of all cities. This set is used to keep track of unvisited cities during the traversal.

4. **`current_city = 0`**: Initializes the current city to start the traversal from city 0.

5. **`tour = [current_city]`**: Initializes a list called `tour` with the starting city.

6. **`total_distance = 0`**: Initializes a variable called `total_distance` to keep track of the total distance traveled.

7. **`while unvisited_cities:`**: Initiates a loop that continues until all cities have been visited (i.e., `unvisited_cities` is not empty).

8. **`nearest_city = min(unvisited_cities, key=lambda city: distances[current_city][city])`**: Finds the nearest unvisited city to the current city by calculating the minimum distance from the current city to each unvisited city using a lambda function as the key for the `min` function.

9. **`total_distance += distances[current_city][nearest_city]`**: Adds the distance from the current city to the nearest city to the total distance traveled.

10. **`current_city = nearest_city`**: Updates the current city to the nearest city.

11. **`unvisited_cities.remove(nearest_city)`**: Removes the nearest city from the set of unvisited cities.

12. **`tour.append(current_city)`**: Appends the nearest city to the tour.

13. **`total_distance += distances[current_city][0]`**: Adds the distance from the current city back to the starting city to the total distance traveled to complete the tour.

14. **`tour.append(0)`**: Appends the starting city (city 0) to the end of the tour to complete the loop.

15. **`return tour, total_distance`**: Returns the tour (list of cities visited in order) and the total distance traveled.

16. **Example usage**: Defines a `distances` matrix representing distances between cities, calls the `tsp_nearest_neighbor` function with this matrix, and prints the resulting tour and total distance traveled.

7. Implementation of the problem solving strategies: either using Forward Chaining or Backward Chaining

```
database = ["Croaks", "Eat Flies", "Shrimps", "Sings"]
```

```
knowbase = ["Frog", "Canary", "Green", "Yellow"]

def display():
   print("\n X is \n1..Croaks \n2.Eat Flies \n3.shrimps \n4.Sings ", end='')
   print("\n Select One ", end='')

def main():
   print("*-----Forward--Chaining-----*", end='')
   display()
   x = int(input())
   print(" \n", end='')
   if x == 1 or x == 2:
      print(" Chance Of Frog ", end='')
   elif x == 3 or x == 4:
      print(" Chance of Canary ", end='')
   else:
      print("\n-------In Valid Option Select --------", end='')

   if x >= 1 and x <= 4:
      print("\n X is ", end='')
      print(database[x-1], end='')
      print("\n Color Is 1.Green 2.Yellow", end='')
      print("\n Select Option ", end='')
      k = int(input())
      if k == 1 and (x == 1 or x == 2):  # frog0 and green1
         print(" yes it is ", end='')
         print(knowbase[0], end='')
         print(" And Color Is ", end='')
         print(knowbase[2], end='')
      elif k == 2 and (x == 3 or x == 4):  # canary1 and yellow3
         print(" yes it is ", end='')
         print(knowbase[1], end='')
         print(" And Color Is ", end='')
         print(knowbase[3], end='')
      else:
         print("\n---InValid Knowledge Database", end='')

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

Explanation

Sure, let's break down the provided code line by line:

1. **`database = ["Croaks", "Eat Flies", "Shrimps", "Sings"]`**: Initializes a list called `database` containing strings representing different attributes.

2. **`knowbase = ["Frog", "Canary", "Green", "Yellow"]`**: Initializes a list called `knowbase` containing strings representing different entities.

3. **`def display():`**: Defines a function named `display()`.

4. **`print("\n X is \n1..Croaks \n2.Eat Flies \n3.shrimps \n4.Sings ", end='')`**: Prints a message prompting the user to select an option, displaying a list of attributes.

5. **`print("\n Select One ", end='')`**: Prints a message asking the user to select one of the options.

6. **`def main():`**: Defines a function named `main()`.

7. **`print("*-----Forward--Chaining-----*", end='')`**: Prints a message indicating the forward chaining process.

8. **`display()`**: Calls the `display()` function to show the options to the user.

9. **`x = int(input())`**: Reads user input as an integer and assigns it to the variable `x`.

10. **`if x == 1 or x == 2:`**: Checks if the user input corresponds to attributes Croaks or Eat Flies.

11. **`print(" Chance Of Frog ", end='')`**: Prints a message indicating the chance of the entity being a Frog.

12. **`elif x == 3 or x == 4:`**: Checks if the user input corresponds to attributes Shrimps or Sings.

13. **`print(" Chance of Canary ", end='')`**: Prints a message indicating the chance of the entity being a Canary.

14. **`else:`**: Executes if the user input does not match any of the above conditions.

15. **`print("\n-------In Valid Option Select --------", end='')`**: Prints a message indicating that the user input is invalid.

16. **`if x >= 1 and x <= 4:`**: Checks if the user input is within the valid range.

17. **`print("\n X is ", end='')`**: Prints a message indicating the selected attribute.

18. **`print(database[x-1], end='')`**: Prints the corresponding attribute from the `database` list based on the user input.

19. **`print("\n Color Is 1.Green 2.Yellow", end='')`**: Prints a message asking the user to select a color.

20. **`print("\n Select Option ", end='')`**: Prints a message asking the user to select an option.

21. **`k = int(input())`**: Reads user input as an integer and assigns it to the variable `k`.

22. **`if k == 1 and (x == 1 or x == 2):`**: Checks if the user input corresponds to selecting the color Green for attributes Croaks or Eat Flies.

23. **`print(" yes it is ", end='')`**: Prints a message indicating that the selected entity matches the attribute.

24. **`print(knowbase[0], end='')`**: Prints the corresponding entity from the `knowbase` list based on the user input.

25. **`print(" And Color Is ", end='')`**: Prints a message indicating the color of the entity.

26. **`print(knowbase[2], end='')`**: Prints the corresponding color from the `knowbase` list based on the user input.

27. **`elif k == 2 and (x == 3 or x == 4):`**: Checks if the user input corresponds to selecting the color Yellow for attributes Shrimps or Sings.

28. **`print(" yes it is ", end='')`**: Prints a message indicating that the selected entity matches the attribute.

29. **`print(knowbase[1], end='')`**: Prints the corresponding entity from the `knowbase` list based on the user input.

30. **`print(" And Color Is ", end='')`**: Prints a message indicating the color of the entity.

31. **`print(knowbase[3], end='')`**: Prints the corresponding color from the `knowbase` list based on the user input.

32. **`else:`**: Executes if the user input does not match any of the above conditions.

33. **`print("\n---InValid Knowledge Database", end='')`**: Prints a message indicating that the user input is invalid.

34. **`if __name__ == "__main__":`**: Checks if the script is being run directly.

35. **`main()`**: Calls the `main()` function to start the program execution.

OUTPUT

-----Forward--Chaining-----
X is
1..Croaks
2.Eat Flies
3.shrimps
4.Sings
Select One 1

Chance Of Frog
X is Croaks
Color Is 1.Green 2.Yellow
Select Option 1
yes it is Frog And Color Is Green

Backward chaining

database = ["Croaks", "Eat Flies", "Shrimps", "Sings"]

```
knowbase = ["Frog", "Canary"]
color = ["Green", "Yellow"]

def display():
   print("\n X is \n1.frog \n2.canary ", end='')
   print("\n Select One ", end='')

def main():
   print("*-----Backward--Chaining-----*", end='')
   display()
   x = int(input())
   print(" \n", end='')
   if x == 1:
      print(" Chance Of eating flies ", end='')
   elif x == 2:
      print(" Chance of shrimping ", end='')
   else:
      print("\n-------In Valid Option Select --------", end='')

   if x >= 1 and x <= 2:
      print("\n X is ", end='')
      print(knowbase[x-1], end='')
      print("\n1.green \n2.yellow")
      k = int(input())
      if k == 1 and x == 1: # frog0 and green1
         print(" yes it is in ", end='')
         print(color[0], end='')
         print(" colour and will ", end='')
         print(database[0])
      elif k == 2 and x == 2: # canary1 and yellow3
         print(" yes it is in", end='')
         print(color[1], end='')
         print(" Colour and will ", end='')
         print(database[1])
      else:
         print("\n---InValid Knowledge Database", end='')

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

OUTPUT

```
*-----Backward--Chaining-----*
```

X is
1.frog
2.canary
Select One 1

Chance Of eating flies
X is Frog
1.green
2.yellow
1
yes it is in Green colour and will Croaks

8. Implement resolution principle on FOPL related problems

```
def negate_literal(literal):
    """
    Negates a literal.
    """
    if literal[0] == '~':
        return literal[1:]
    else:
        return '~' + literal

def resolve(clause1, clause2):
    """
    Resolves two clauses.
    """
    for literal1 in clause1:
        for literal2 in clause2:
            if literal1 == negate_literal(literal2):
                resolvent = [lit for lit in (clause1 + clause2) if lit != literal1 and lit != literal2]
                return resolvent
    return None

def resolution(clauses):
    """
    Applies resolution on a list of clauses until no new clauses can be derived.
    """
    new_clauses = set(map(tuple, clauses))  # Convert lists to tuples
    while True:
        n = len(new_clauses)
        pairs = [(c1, c2) for c1 in new_clauses for c2 in new_clauses if c1 != c2]
        for (c1, c2) in pairs:
            resolvent = resolve(c1, c2)
            if resolvent is None:
                continue
            if not resolvent:  # Empty clause
                return True
            new_clauses.add(tuple(resolvent))
        if len(new_clauses) == n:
            return False

# Example
clauses = [
    ['~P', 'Q'],
    ['~Q', 'R'],
    ['P', 'S'],
    ['~R', 'S'],
    ['~P', 'R']
]
```

```
if resolution(clauses):
    print("The given statement is entailed.")
else:
    print("The given statement is not entailed.")
```

Output

```
The given statement is not entailed.

=== Code Execution Successful ===
```

Explanation

Sure, let's break down the provided code line by line:

1. **`def negate_literal(literal):`**: Defines a function named `negate_literal` that takes a `literal` as input and returns its negation.

2. **`if literal[0] == '~':`**: Checks if the first character of the `literal` is a tilde (`~`), indicating that the literal is negated.

3. **`return literal[1:]`**: If the `literal` is already negated, it returns the literal without the tilde, effectively negating it. For example, if the literal is `'~P'`, it returns `'P'`.

4. **`else:`**: Executes if the `literal` is not negated.

5. **`return '~' + literal`**: Negates the `literal` by prepending a tilde to it. For example, if the literal is `'P'`, it returns `'~P'`.

6. **`def resolve(clause1, clause2):`**: Defines a function named `resolve` that takes two clauses (`clause1` and `clause2`) as input and tries to resolve them.

7. **`for literal1 in clause1:`**: Iterates over each literal in `clause1`.

8. **`for literal2 in clause2:`**: Iterates over each literal in `clause2`.

9. **`if literal1 == negate_literal(literal2):`**: Checks if `literal1` is the negation of `literal2`. If so, they can be resolved.

10. **`resolvent = [lit for lit in (clause1 + clause2) if lit != literal1 and lit != literal2]`**: Creates the resolvent clause by combining `clause1` and `clause2` and removing the literals that resolved.

11. **`return resolvent`**: Returns the resolvent clause.

12. **`return None`**: Returns `None` if no resolution is possible between the clauses.

13. **`def resolution(clauses):`**: Defines a function named `resolution` that takes a list of clauses (`clauses`) as input and applies resolution until no new clauses can be derived.

14. **`new_clauses = set(map(tuple, clauses))`**: Converts the list of clauses into a set of tuples. This is done to ensure uniqueness and avoid duplicate clauses during resolution.

15. **`while True:`**: Initiates an infinite loop to apply resolution until no new clauses can be derived.

16. **`n = len(new_clauses)`**: Records the length of `new_clauses` to track if any new clauses are derived in the current iteration.

17. **`pairs = [(c1, c2) for c1 in new_clauses for c2 in new_clauses if c1 != c2]`**: Generates all possible pairs of clauses from `new_clauses`, excluding pairs with identical clauses.

18. **`for (c1, c2) in pairs:`**: Iterates over each pair of clauses.

19. **`resolvent = resolve(c1, c2)`**: Attempts to resolve the current pair of clauses.

20. **`if resolvent is None:`**: Checks if resolution is not possible for the current pair of clauses.

21. **`if not resolvent:`**: Checks if the resolvent is an empty clause, indicating a contradiction.

22. **`new_clauses.add(tuple(resolvent))`**: Adds the resolvent clause to `new_clauses`.

23. **`if len(new_clauses) == n:`**: Checks if no new clauses were derived in the current iteration.

24. **`return False`**: Returns `False` to indicate that the given statement is not entailed.

25. **`return True`**: Returns `True` to indicate that the given statement is entailed if an empty clause is derived.

26. **`clauses = [...]`**: Defines a list of clauses representing a set of logical statements.

27. **`if resolution(clauses):`**: Calls the `resolution` function with the list of clauses as input and checks if the given statement is entailed.

28. **`print("The given statement is entailed.")`**: Prints a message indicating that the given statement is entailed if the resolution succeeds.

29. **`else:`**: Executes if the resolution fails to derive an empty clause.

30. **`print("The given statement is not entailed.")`**: Prints a message indicating that the given statement is not entailed.

9. Implement Tic-Tac-Toe game using Python

```
import os
import time

board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
player = 1

# Win Flags
Win = 1
Draw = -1
Running = 0
Stop = 1

Game = Running
Mark = 'X'

def DrawBoard():
    print(" %c | %c | %c " % (board[1], board[2], board[3]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[4], board[5], board[6]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[7], board[8], board[9]))
    print("   |   |   ")

def CheckPosition(x):
    if board[x] == ' ':
        return True
    else:
        return False

def CheckWin():
    global Game

    if board[1] == board[2] and board[2] == board[3] and board[1] != ' ':
        Game = Win
    elif board[4] == board[5] and board[5] == board[6] and board[4] != ' ':
        Game = Win
    elif board[7] == board[8] and board[8] == board[9] and board[7] != ' ':
        Game = Win
    elif board[1] == board[4] and board[4] == board[7] and board[1] != ' ':
        Game = Win
    elif board[2] == board[5] and board[5] == board[8] and board[2] != ' ':
        Game = Win
    elif board[3] == board[6] and board[6] == board[9] and board[3] != ' ':
        Game = Win
    elif board[1] == board[5] and board[5] == board[9] and board[5] != ' ':
        Game = Win
    elif board[3] == board[5] and board[5] == board[7] and board[5] != ' ':
        Game = Win
```

```
    elif board[1] != ' ' and board[2] != ' ' and board[3] != ' ' and \
            board[4] != ' ' and board[5] != ' ' and board[6] != ' ' and \
            board[7] != ' ' and board[8] != ' ' and board[9] != ' ':
        Game = Draw
    else:
        Game = Running


print("Player 1 [X] --- Player 2 [O]\n")
print()
print()
print("Please Wait...")
time.sleep(3)

while Game == Running:
    os.system('cls')
    DrawBoard()

    if player % 2 != 0:
        print("Player 1's chance")
        Mark = 'X'
    else:
        print("Player 2's chance")
        Mark = 'O'

    choice = int(input("Enter the position between [1-9] where you want to mark: "))
    if CheckPosition(choice):
        board[choice] = Mark
        player += 1
        CheckWin()

    os.system('cls')
    DrawBoard()

    if Game == Draw:
        print("Game Draw")
    elif Game == Win:
        player -= 1
        if player % 2 != 0:
            print("Player 1 Won")
        else:
            print("Player 2 Won")
```

OUTPUT

```
   |   |
 X |   |
___|___|___
 O | X |
___|___|___
   |   |
   |   |
Player 2's chance
Enter the position between [1-9] where you want to mark: 9
 X |   |
___|___|___
 O | X |
___|___|___
   |   | O
   |   |
 X |   |
___|___|___
 O | X |
___|___|___
   |   | O
   |   |
Player 1's chance
Enter the position between [1-9] where you want to mark: 3
 X |   | X
___|___|___
 O | X |
___|___|___
   |   | O
   |   |
 X |   | X
___|___|___
 O | X |
___|___|___
   |   | O
   |   |
Player 2's chance
Enter the position between [1-9] where you want to mark: 2
 X | O | X
___|___|___
 O | X |
___|___|___
   |   | O
   |   |
 X | O | X
___|___|___
 O | X |
___|___|___
   |   | O
   |   |
Player 1's chance
Enter the position between [1-9] where you want to mark: 7
 X | O | X
___|___|___
 O | X |
___|___|___
 X |   | O
   |   |
Player 1 Won
```

Explanation

1. **`import os`**: Imports the `os` module, which provides a portable way to interact with the operating system.

2. **`import time`**: Imports the `time` module, which provides various time-related functions.

3. **`board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']`**: Initializes a list named `board` with 10 elements, representing the Tic-Tac-Toe board.

4. **`player = 1`**: Initializes a variable named `player` to keep track of the current player.

5. **`Win = 1`, `Draw = -1`, `Running = 0`, `Stop = 1`**: Defines constants representing different game states.

6. **`Game = Running`**: Initializes the game state to `Running`.

7. **`Mark = 'X'`**: Initializes the variable `Mark` to ``'X'`, indicating the mark (either 'X' or 'O') currently being played.

8. **`def DrawBoard():`**: Defines a function named `DrawBoard()` to draw the Tic-Tac-Toe board.

9. **`def CheckPosition(x):`**: Defines a function named `CheckPosition(x)` to check if a position on the board is empty.

10. **`def CheckWin():`**: Defines a function named `CheckWin()` to check if a player has won the game.

11. **`print("Player 1 [X] --- Player 2 [O]\n")`**: Prints the initial message indicating which player is assigned 'X' and which player is assigned 'O'.

12. **`while Game == Running:`**: Starts a loop that continues until the game state changes to something other than `Running`.

13. **`os.system('cls')`**: Clears the console screen using the `cls` command (for Windows). This is done to refresh the screen after each move.

14. **`if player % 2 != 0:`** and **`else:`**: Checks which player's turn it is based on the value of `player`.

15. **`choice = int(input("Enter the position between [1-9] where you want to mark: "))`**: Prompts the current player to enter a position on the board where they want to mark 'X' or 'O'.

16. **`if CheckPosition(choice):`**: Checks if the chosen position is valid (i.e., empty) using the `CheckPosition()` function.

17. **`board[choice] = Mark`**: Marks the chosen position on the board with the current player's mark.

18. **`CheckWin()`**: Checks if the game has been won or drawn after the current move.

19. **`os.system('cls')`**: Clears the console screen again to refresh the display.

20. **`if Game == Draw:`** and **`elif Game == Win:`**: Checks if the game has ended in a draw or if a player has won.

21. **`player -= 1`**: Adjusts the value of `player` to correctly identify the winning player.

22. **`print("Player 1 Won")`** and **`print("Player 2 Won")`**: Prints the message indicating which player has won the game.

Overall, this code implements a simple Tic-Tac-Toe game where two players take turns marking positions on a 3x3 board, and the game ends when one player wins or when the board is filled without a winner (resulting in a draw).

10. **Build a bot which provides all the information related to text in search box**

```
import requests
from bs4 import BeautifulSoup

def search_wikipedia(query):
    try:
        # Construct the search URL
        search_url = f"https://en.wikipedia.org/wiki/{query.replace(' ', '_')}"

        # Send a GET request to the search URL
        response = requests.get(search_url)
        response.raise_for_status()  # Raise an exception for any HTTP errors

        # Parse the HTML content of the page
        soup = BeautifulSoup(response.text, 'html.parser')

        # Extract the summary paragraph
        summary_paragraph = soup.find('title').get_text()

        return summary_paragraph

    except requests.exceptions.RequestException as e:
        return f"Error occurred while fetching data: {e}"

def main():
    print("Welcome to the Text Information Bot!")
    while True:
        query = input("Enter your search query (type 'exit' to quit): ").strip()
        if query.lower() == 'exit':
            print("Thank you for using the Text Information Bot. Goodbye!")
            break

        # Search Wikipedia for the query
        result = search_wikipedia(query)
        print(result)
        print()

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

OUTPUT

```
Welcome to the Text Information Bot!
Enter your search query (type 'exit' to quit): John F Kennedy
John F. Kennedy - Wikipedia

Enter your search query (type 'exit' to quit):
```

Explanation

1. **`import requests`**: Imports the `requests` module, which allows sending HTTP requests easily.

2. **`from bs4 import BeautifulSoup`**: Imports the `BeautifulSoup` class from the `bs4` module, which is used for web scraping and parsing HTML.

3. **`def search_wikipedia(query):`**: Defines a function named `search_wikipedia` that takes a search query as input and returns the summary paragraph from Wikipedia related to the query.

4. **`try:`** and **`except requests.exceptions.RequestException as e:`**: These lines handle any potential exceptions that may occur during the HTTP request process, such as network errors or invalid URLs.

5. **`search_url = f"https://en.wikipedia.org/wiki/{query.replace(' ', '_')}"`**: Constructs the URL for the Wikipedia search based on the input query. It replaces spaces in the query with underscores to form a valid URL.

6. **`response = requests.get(search_url)`**: Sends a GET request to the constructed search URL and stores the response.

7. **`response.raise_for_status()`**: Raises an exception if the HTTP response status code indicates an error (e.g., 404 Not Found).

8. **`soup = BeautifulSoup(response.text, 'html.parser')`**: Creates a BeautifulSoup object `soup` by parsing the HTML content of the response text.

9. **`summary_paragraph = soup.find('title').get_text()`**: Finds the title tag in the HTML content and extracts the text of the title, which typically represents the summary or topic of the Wikipedia page.

10. **`return summary_paragraph`**: Returns the extracted summary paragraph.

11. **`def main():`**: Defines the main function of the program.

12. **`print("Welcome to the Text Information Bot!")`**: Prints a welcome message to the user.

13. **`while True:`**: Starts an infinite loop to continuously accept user input until the user decides to exit.

14. **`query = input("Enter your search query (type 'exit' to quit): ").strip()`**: Prompts the user to enter a search query and strips any leading or trailing whitespace from the input.

15. **`if query.lower() == 'exit':`**: Checks if the user wants to exit the program.

16. **`result = search_wikipedia(query)`**: Calls the `search_wikipedia` function to search Wikipedia for the user's query and stores the result.

17. **`print(result)`**: Prints the result, which is the summary paragraph retrieved from Wikipedia.

18. **`if __name__ == "__main__":`**: Checks if the script is being run directly (not imported as a module).

19. **`main()`**: Calls the `main` function to start the program execution.

Overall, this code creates a simple text-based bot that allows users to search for information on Wikipedia by entering a query. It retrieves the summary paragraph from the corresponding Wikipedia page and displays it to the user.

11. Implement any Game and demonstrate the Game playing strategies

```
import os
import time
```

```
board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
player = 1

# Win Flags
Win = 1
Draw = -1
Running = 0
Stop = 1

Game = Running
Mark = 'X'

def DrawBoard():
    print(" %c | %c | %c " % (board[1], board[2], board[3]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[4], board[5], board[6]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[7], board[8], board[9]))
    print("   |   |   ")

def CheckPosition(x):
    if board[x] == ' ':
        return True
    else:
        return False

def CheckWin():
    global Game

    if board[1] == board[2] and board[2] == board[3] and board[1] != ' ':
        Game = Win
    elif board[4] == board[5] and board[5] == board[6] and board[4] != ' ':
        Game = Win
    elif board[7] == board[8] and board[8] == board[9] and board[7] != ' ':
        Game = Win
    elif board[1] == board[4] and board[4] == board[7] and board[1] != ' ':
        Game = Win
    elif board[2] == board[5] and board[5] == board[8] and board[2] != ' ':
        Game = Win
    elif board[3] == board[6] and board[6] == board[9] and board[3] != ' ':
        Game = Win
    elif board[1] == board[5] and board[5] == board[9] and board[5] != ' ':
        Game = Win
    elif board[3] == board[5] and board[5] == board[7] and board[5] != ' ':
        Game = Win
    elif board[1] != ' ' and board[2] != ' ' and board[3] != ' ' and \
            board[4] != ' ' and board[5] != ' ' and board[6] != ' ' and \
            board[7] != ' ' and board[8] != ' ' and board[9] != ' ':
```

```
            Game = Draw
        else:
            Game = Running


print("Player 1 [X] --- Player 2 [O]\n")
print()
print()
print("Please Wait...")
time.sleep(3)

while Game == Running:
    os.system('cls')
    DrawBoard()

    if player % 2 != 0:
        print("Player 1's chance")
        Mark = 'X'
    else:
        print("Player 2's chance")
        Mark = 'O'

    choice = int(input("Enter the position between [1-9] where you want to mark: "))
    if CheckPosition(choice):
        board[choice] = Mark
        player += 1
        CheckWin()

    os.system('cls')
    DrawBoard()

    if Game == Draw:
        print("Game Draw")
    elif Game == Win:
        player -= 1
        if player % 2 != 0:
            print("Player 1 Won")
        else:
            print("Player 2 Won")
```

OUTPUT

```
     |     |
  X  |     |
_____|_____|_____
  O  |  X  |
_____|_____|_____
```

```
   |   |
   |   |
Player 2's chance
Enter the position between [1-9] where you want to mark: 9
 X |   |
___|___|___
 O | X |
___|___|___
   |   | O
   |   |
 X |   |
___|___|___
 O | X |
___|___|___
   |   | O
   |   |
Player 1's chance
Enter the position between [1-9] where you want to mark: 3
 X |   | X
___|___|___
 O | X |
___|___|___
   |   | O
   |   |
 X |   | X
___|___|___
 O | X |
___|___|___
   |   | O
   |   |
Player 2's chance
Enter the position between [1-9] where you want to mark: 2
 X | O | X
___|___|___
 O | X |
___|___|___
   |   | O
   |   |
 X | O | X
___|___|___
 O | X |
___|___|___
   |   | O
   |   |
Player 1's chance
Enter the position between [1-9] where you want to mark: 7
 X | O | X
___|___|___
 O | X |
___|___|___
 X |   | O
   |   |
Player 1 Won
```

Explanation

1. **`import os`**: Imports the `os` module, which provides a portable way to interact with the operating system.

2. **`import time`**: Imports the `time` module, which provides various time-related functions.

3. **`board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']`**: Initializes a list named `board` with 10 elements, representing the Tic-Tac-Toe board.

4. **`player = 1`**: Initializes a variable named `player` to keep track of the current player.

5. **`Win = 1`, `Draw = -1`, `Running = 0`, `Stop = 1`**: Defines constants representing different game states.

6. **`Game = Running`**: Initializes the game state to `Running`.

7. **`Mark = 'X'`**: Initializes the variable `Mark` to `'X'`, indicating the mark (either 'X' or 'O') currently being played.

8. **`def DrawBoard():`**: Defines a function named `DrawBoard()` to draw the Tic-Tac-Toe board.

9. **`def CheckPosition(x):`**: Defines a function named `CheckPosition(x)` to check if a position on the board is empty.

10. **`def CheckWin():`**: Defines a function named `CheckWin()` to check if a player has won the game.

11. **`print("Player 1 [X] --- Player 2 [O]\n")`**: Prints the initial message indicating which player is assigned 'X' and which player is assigned 'O'.

12. **`while Game == Running:`**: Starts a loop that continues until the game state changes to something other than `Running`.

13. **`os.system('cls')`**: Clears the console screen using the `cls` command (for Windows). This is done to refresh the screen after each move.

14. **`if player % 2 != 0:`** and **`else:`**: Checks which player's turn it is based on the value of `player`.

15. **`choice = int(input("Enter the position between [1-9] where you want to mark: "))`**: Prompts the current player to enter a position on the board where they want to mark 'X' or 'O'.

16. **`if CheckPosition(choice):`**: Checks if the chosen position is valid (i.e., empty) using the `CheckPosition()` function.

17. **`board[choice] = Mark`**: Marks the chosen position on the board with the current player's mark.

18. **`CheckWin()`**: Checks if the game has been won or drawn after the current move.

19. **`os.system('cls')`**: Clears the console screen again to refresh the display.

20. **`if Game == Draw:`** and **`elif Game == Win:`**: Checks if the game has ended in a draw or if a player has won.

21. **`player -= 1`**: Adjusts the value of `player` to correctly identify the winning player.

22. **`print("Player 1 Won")`** and **`print("Player 2 Won")`**: Prints the message indicating which player has won the game.

Overall, this code implements a simple Tic-Tac-Toe game where two players take turns marking positions on a 3x3 board, and the game ends when one player wins or when the board is filled without a winner (resulting in a draw).