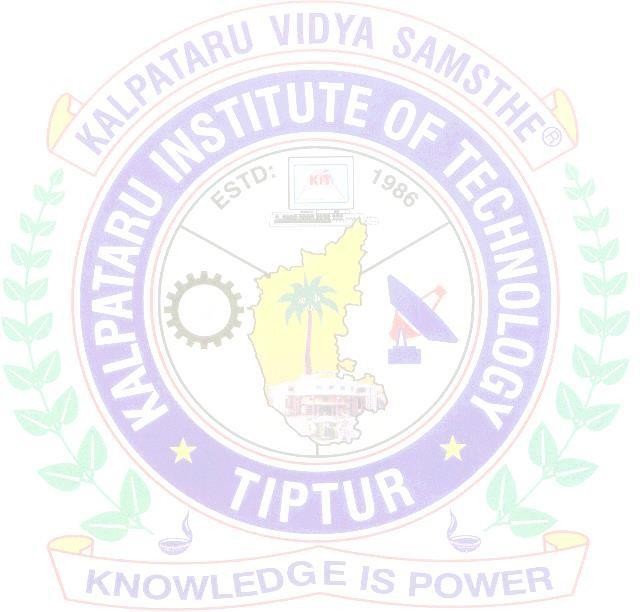


Visvesvaraya Technological University

“Jnana Sangama”, Belagavi-590018

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Fourth Semester B.E.



[As per Choice Based Credit System (CBCS) Scheme]

(For Internal Circulation Only)

**Integrated Professional Core Course (IPCC)**

**Artificial Intelligence (BAD402)**

**Lab Manual**

(For Reference Only)

|  |  |
| --- | --- |
| Name |  |
| USN |  |
| Section |  |
| Lab Batch |  |
| Day /Time |  |

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Kalpataru Institute of Technology, Tiptur - 572 201

Department of Artificial Intelligence and Machine Learning

AY: 2024-2025

**VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELAGAVI**

# 

# Integrated Professional Core Course (IPCC)

# ARTIFICIAL INTELLIGENCE LAB MANUAL

**BAD402**

# IV Semester



**KALPATARU INSTITUTE OF TECHNOLOGY, TIPTUR DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING**

**2024-25**

**CONTENTS**

|  |  |  |
| --- | --- | --- |
| **PROGRAM NO.** | **PROGRAM TITLE** | **PAGE NO.** |
| 1 | Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem | 1-3 |
| 2 | Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems using Python | 4-8 |
| 3 | Implement A\* Search algorithm | 9-11 |
| 4 | Implement AO\* Search algorithm | 12-13 |
| 5 | Solve 8-Queens Problem with suitable assumptions | 14-15 |
| 6 | Implementation of TSP using heuristic approach | 16 |
| 7 | Implementation of the problem solving strategies: either using Forward Chaining or Backward Chaining | 17-18 |
| 8 | Implement resolution principle on FOPL related problems | 19-20 |
| 9 | Implement Tic-Tac-Toe game using Python | 21-23 |
| 10 | Build a bot which provides all the information related to text in search box | 24 |
| 11 | Implement any Game and demonstrate the Game playing strategies | 25 |

**KALPATARU INSTITUTE OF TECHNOLOGY**

**(Accredited by NBA, Approved by A.I.C.T.E. New Delhi, Recognized by Govt. of Karnataka & Affiliated to V.T U., Belagavi)**

## DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

**Vision and Mission of the Institution**

# Vision

“To bring forth technical graduates of high caliber with a strong character and to uphold the spiritual and cultural values of our country.”

# Mission

“To impart quality technical and managerial education at graduate and post graduate levels through our dedicated and well qualified faculty.”

**Vision and Mission of the AI & ML Department**

# Vision

“To create a community of AI and ML specialists distinguished by their technical excellence and moral principles, who champion responsible innovation and honor our country's spiritual and cultural traditions, advancing technology for the betterment of society.”

# Mission

**M1:** “To advance AI and ML through ground-breaking development, encouraging students and faculty to pursue innovative solutions that address global challenges and drive societal progress.”  
**M2:** “To provide a supportive and inclusive environment that nurtures the intellectual and personal growth of our students, preparing them to become leaders in AI&ML.”  
**M3:** “To integrate the nation's spiritual and cultural values into the AI and ML curriculum, fostering respect for our heritage and guiding students to consider the cultural impacts of their innovations.”

**KALPATARU INSTITUTE OF TECHNOLOGY**

**(Accredited by NBA, Approved by A.I.C.T.E. New Delhi, Recognized by Govt. of Karnataka & Affiliated to V.T U., Belagavi)**

## DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

|  |  |
| --- | --- |
| **Program Outcomes** | |
| **a.** | **Engineering Knowledge:** Apply knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems. |
| **b.** | **Problem Analysis: Identify,** formulate, research literature and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences |
| **c.** | **Design/ Development of Solutions**: Design solutions for complex engineering problems and design system components or processes that meet specified needs with appropriate consideration for public health and safety, cultural, societal and environmental considerations. |
| **d.** | **Conduct investigations of complex problems** using research-based knowledge and research methods including design of experiments, analysis and interpretation of data and synthesis of information to provide valid conclusions. |
| **e.** | **Modern Tool Usage**: Create, select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modeling to Complex engineering activities with an under- standing of the limitations. |
| **f.** | **The Engineer and Society**: Apply reasoning informed by contextual knowledge to assess societal, health, safety, legal and cultural issues and the Consequent responsibilities relevant to professional engineering practice. |
| **g.** | **Environment and Sustainability**: Understand the impact of professional Engineering solutions in societal and environmental contexts and demonstrate knowledge of and need for sustainable development. |
| **h.** | **Ethics**: Apply ethical principles and commit to professional ethics and Responsibilities and norms of engineering practice. |
| **i.** | **Individual and Team Work:** Function effectively as an individual, and as a member or leader in diverse teams and in multi disciplinary settings. |
| **j.** | **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**.** |
| **k.** | **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. |
| **l.** | **Project Management and Finance:** Demonstrate knowledge and understanding of 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. |
| **Program Specific Outcomes** | |
| **m.** | **PSO1**: The ability to comprehend, analyse, and apply knowledge of human cognition, Artificial Intelligence, Machine Learning, and data engineering to real-world problems, addressing future challenges. |
| **n.** | **PSO2**: The ability to cultivate computational knowledge and project development skills, utilizing innovative tools and techniques to address problems in Deep Learning, Machine Learning, and Artificial Intelligence. |

**Program 1**

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

**code**

class WaterJugState:

def \_\_init\_\_(self, jug1, jug2):

self.jug1 = jug1

self.jug2 = jug2

def \_\_eq\_\_(self, other):

return self.jug1 == other.jug1 and self.jug2 == other.jug2

def \_\_hash\_\_(self):

return hash((self.jug1, self.jug2))

def dfs(current\_state, visited, jug1\_capacity, jug2\_capacity, target\_volume):

if current\_state.jug1 == target\_volume or current\_state.jug2 == target\_volume:

if current\_state.jug1 == target\_volume :

print("Jug 1 now has", target\_volume, "liters.")

else:

print("Jug 2 now has", target\_volume, "liters.")

return True

visited.add(current\_state)

# Define all possible operations: (action, from\_jug, to\_jug)

operations = [

('Fill Jug 1', jug1\_capacity, current\_state.jug2),

('Fill Jug 2', current\_state.jug1, jug2\_capacity),

('Empty Jug 1', 0, current\_state.jug2),

('Empty Jug 2', current\_state.jug1, 0),

('Pour Jug 1 to Jug 2',

max(0, current\_state.jug1 + current\_state.jug2 - jug2\_capacity),

min(jug2\_capacity, current\_state.jug1 + current\_state.jug2)),

('Pour Jug 2 to Jug 1',

min(jug1\_capacity, current\_state.jug1 + current\_state.jug2),

max(0, current\_state.jug1 + current\_state.jug2 - jug1\_capacity))

]

for operation in operations:

action, new\_jug1, new\_jug2 = operation

new\_state = WaterJugState(new\_jug1, new\_jug2)

if new\_state not in visited:

print(f"Trying: {action} => ({new\_jug1}, {new\_jug2})")

if dfs(new\_state, visited, jug1\_capacity, jug2\_capacity, target\_volume):

return True

# print(new\_jug1, new\_jug2)

return False

def solve\_water\_jug\_problem(jug1\_capacity, jug2\_capacity, target\_volume):

initial\_state = WaterJugState(0, 0)

visited = set()

if dfs(initial\_state, visited, jug1\_capacity, jug2\_capacity, target\_volume):

print("Solution found!")

# print(jug1\_capacity, jug2\_capacity)

else:

print("Solution not possible.")

# Example usage:

jug1\_capacity = int(input("Enter Jug 1 capacity : "))

jug2\_capacity = int(input("Enter Jug 1 capacity : "))

target\_volume = int(input("Enter Target Volume : "))

print(f"Solving Water Jug Problem with capacities ({jug1\_capacity}, {jug2\_capacity}) to measure {target\_volume} liters.")

solve\_water\_jug\_problem(jug1\_capacity, jug2\_capacity, target\_volume)

$ python 01\_DFS\_WJP.py

Enter Jug 1 capacity : 5

Enter Jug 1 capacity : 4

Enter Target Volume : 3

Solving Water Jug Problem with capacities (5, 4) to measure 3 liters.

Trying: Fill Jug 1 => (5, 0)

Trying: Fill Jug 2 => (5, 4)

Trying: Empty Jug 1 => (0, 4)

Trying: Pour Jug 2 to Jug 1 => (4, 0)

Trying: Fill Jug 2 => (4, 4)

Trying: Pour Jug 2 to Jug 1 => (5, 3)

Jug 2 now has 3 liters.

Solution found!

$ python 01\_DFS\_WJP.py

Enter Jug 1 capacity : 2

Enter Jug 1 capacity : 2

Enter Target Volume : 6

Solving Water Jug Problem with capacities (2, 2) to measure 6 liters.

Trying: Fill Jug 1 => (2, 0)

Trying: Fill Jug 2 => (2, 2)

Trying: Empty Jug 1 => (0, 2)

Solution not possible.

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

from queue import PriorityQueue

# Define the state class for the Missionaries and Cannibals Problem

class State:

def \_\_init\_\_(self, left\_m, left\_c, boat, right\_m, right\_c):

self.left\_m = left\_m # Number of missionaries on the left bank

self.left\_c = left\_c # Number of cannibals on the left bank

self.boat = boat # 1 if boat is on the left bank, 0 if on the right bank

self.right\_m = right\_m # Number of missionaries on the right bank

self.right\_c = right\_c # Number of cannibals on the right bank

def is\_valid(self):

# Check if the state is valid (no missionaries eaten on either bank)

if self.left\_m < 0 or self.left\_c < 0 or self.right\_m < 0 or self.right\_c < 0:

return False

if self.left\_m > 0 and self.left\_c > self.left\_m:

return False

if self.right\_m > 0 and self.right\_c > self.right\_m:

return False

return True

def is\_goal(self):

# Check if the state is the goal state (all missionaries and cannibals on the right bank)

return self.left\_m == 0 and self.left\_c == 0

def \_\_lt\_\_(self, other):

# Define less-than operator for PriorityQueue comparison (used in Best-First Search)

return False

def \_\_eq\_\_(self, other):

# Define equality operator for comparing states

return self.left\_m == other.left\_m and self.left\_c == other.left\_c \

and self.boat == other.boat and self.right\_m == other.right\_m \

and self.right\_c == other.right\_c

def \_\_hash\_\_(self):

# Define hash function for storing states in a set

return hash((self.left\_m, self.left\_c, self.boat, self.right\_m, self.right\_c))

def successors(state):

# Generate all valid successor states from the current state

succ\_states = []

if state.boat == 1: # Boat is on the left bank

for m in range(3):

for c in range(3):

if 1 <= m + c <= 2: # Boat capacity is 2

new\_state = State(state.left\_m - m, state.left\_c - c, 0,

state.right\_m + m, state.right\_c + c)

if new\_state.is\_valid():

succ\_states.append(new\_state)

else: # Boat is on the right bank

for m in range(3):

for c in range(3):

if 1 <= m + c <= 2: # Boat capacity is 2

new\_state = State(state.left\_m + m, state.left\_c + c, 1,

state.right\_m - m, state.right\_c - c)

if new\_state.is\_valid():

succ\_states.append(new\_state)

return succ\_states

def best\_first\_search():

start\_state = State(3, 3, 1, 0, 0)

goal\_state = State(0, 0, 0, 3, 3)

frontier = PriorityQueue()

frontier.put((0, start\_state)) # Priority queue with (cost, state)

came\_from = {}

cost\_so\_far = {}

came\_from[start\_state] = None

cost\_so\_far[start\_state] = 0

while not frontier.empty():

current\_cost, current\_state = frontier.get()

if current\_state == goal\_state:

# Reconstruct the path from start\_state to goal\_state

path = []

while current\_state is not None:

path.append(current\_state)

current\_state = came\_from[current\_state]

path.reverse()

return path

for next\_state in successors(current\_state):

new\_cost = cost\_so\_far[current\_state] + 1 # Uniform cost of 1 for each move

if next\_state not in cost\_so\_far or new\_cost < cost\_so\_far[next\_state]:

cost\_so\_far[next\_state] = new\_cost

priority = new\_cost # Best-First Search uses cost as priority

frontier.put((priority, next\_state))

came\_from[next\_state] = current\_state

return None # No path found

def print\_solution(path):

if path is None:

print("No solution found.")

else:

print("Solution found!")

for i, state in enumerate(path):

print(f"Step {i}:")

print(f"Left Bank: {state.left\_m} missionaries, {state.left\_c} cannibals")

print(f"Boat is {'on the left' if state.boat == 1 else 'on the right'} bank")

print(f"Right Bank: {state.right\_m} missionaries, {state.right\_c} cannibals")

print("------------")

# Main function to run the Best-First Search and print the solution

if \_\_name\_\_ == "\_\_main\_\_":

solution\_path = best\_first\_search()

print\_solution(solution\_path)

**out put**

Solution found!

Step 0:

Left Bank: 3 missionaries, 3 cannibals

Boat is on the left bank

Right Bank: 0 missionaries, 0 cannibals

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

Step 1:

Left Bank: 3 missionaries, 1 cannibals

Boat is on the right bank

Right Bank: 0 missionaries, 2 cannibals

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

Step 2:

Left Bank: 3 missionaries, 2 cannibals

Boat is on the left bank

Right Bank: 0 missionaries, 1 cannibals

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

Step 3:

Left Bank: 3 missionaries, 0 cannibals

Boat is on the right bank

Right Bank: 0 missionaries, 3 cannibals

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

Step 4:

Left Bank: 3 missionaries, 1 cannibals

Boat is on the left bank

Right Bank: 0 missionaries, 2 cannibals

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

Step 5:

Left Bank: 1 missionaries, 1 cannibals

Boat is on the right bank

Right Bank: 2 missionaries, 2 cannibals

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

Step 6:

Left Bank: 2 missionaries, 2 cannibals

Boat is on the left bank

Right Bank: 1 missionaries, 1 cannibals

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

Step 7:

Left Bank: 0 missionaries, 2 cannibals

Boat is on the right bank

Right Bank: 3 missionaries, 1 cannibals

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

Step 8:

Left Bank: 0 missionaries, 3 cannibals

Boat is on the left bank

Right Bank: 3 missionaries, 0 cannibals

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

Step 9:

Left Bank: 0 missionaries, 1 cannibals

Boat is on the right bank

Right Bank: 3 missionaries, 2 cannibals

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

Step 10:

Left Bank: 0 missionaries, 2 cannibals

Boat is on the left bank

Right Bank: 3 missionaries, 1 cannibals

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

Step 11:

Left Bank: 0 missionaries, 0 cannibals

Boat is on the right bank

Right Bank: 3 missionaries, 3 cannibals

**3.Implement A\* Search algorithm.**

import heapq

class Node:

def \_\_init\_\_(self, state, parent=None, action=None, cost=0, heuristic=0):

self.state = state # Current state in the search space

self.parent = parent # Parent node

self.action = action # Action that led to this node from the parent node

self.cost = cost # Cost to reach this node from the start node

self.heuristic = heuristic # Heuristic estimate of the cost to reach the goal

def \_\_lt\_\_(self, other):

return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def parse\_graph\_input():

graph = {}

num\_edges = int(input("Enter the number of edges: "))

for \_ in range(num\_edges):

u, v, cost = input("Enter an edge (format: u v cost): ").split()

cost = int(cost)

if u not in graph:

graph[u] = []

if v not in graph:

graph[v] = []

graph[u].append((v, cost))

graph[v].append((u, cost))

return graph

def astar\_search(start\_state, goal\_test, successors, heuristic):

# Priority queue to store nodes ordered by f = g + h

frontier = []

heapq.heappush(frontier, Node(start\_state, None, None, 0, heuristic(start\_state)))

explored = set()

while frontier:

current\_node = heapq.heappop(frontier)

current\_state = current\_node.state

if goal\_test(current\_state):

# Reconstruct the path from the goal node to the start node

path = []

while current\_node.parent is not None:

path.append((current\_node.action, current\_node.state))

current\_node = current\_node.parent

path.reverse()

return path

explored.add(current\_state)

# Generate successors for the current state using the `successors` function

for action, successor\_state, step\_cost in successors(current\_state):

if successor\_state not in explored:

new\_cost = current\_node.cost + step\_cost

new\_node = Node(successor\_state, current\_node, action, new\_cost, heuristic(successor\_state))

heapq.heappush(frontier, new\_node)

return None # No path found

if \_\_name\_\_ == "\_\_main\_\_":

# Get user input to define the graph

print("Define the graph:")

graph = parse\_graph\_input()

start\_state = input("Enter the start state: ")

goal\_state = input("Enter the goal state: ")

def goal\_test(state):

return state == goal\_state

def successors(state):

# Generate successor states from the current state based on the graph

successors\_list = []

for neighbor, cost in graph.get(state, []):

action = f"Move to {neighbor}" # Default action (e.g., "Move to B")

successor\_state = neighbor

step\_cost = cost

successors\_list.append((action, successor\_state, step\_cost))

return successors\_list

def heuristic(state):

# Define a simple heuristic function (e.g., straight-line distance)

heuristic\_values = {key: abs(ord(key) - ord(goal\_state)) for key in graph.keys()}

return heuristic\_values.get(state, float('inf')) # Default to infinity if state not found

# Perform A\* search using custom successors function

path = astar\_search(start\_state, goal\_test, successors, heuristic)

# Print the resulting path found by A\* search

if path:

print("Path found:")

for action, state in path:

print(f"Action: {action}, State: {state}")

else:

print("No path found.")

**out put**

$ python 03Astar\_search.py

Define the graph:

Enter the number of edges: 7

Enter an edge (format: u v cost): A B 1

Enter an edge (format: u v cost): A C 3

Enter an edge (format: u v cost): B C 1

Enter an edge (format: u v cost): B D 2

Enter an edge (format: u v cost): C D 1

Enter an edge (format: u v cost): D E 4

Enter an edge (format: u v cost): E G 3

Enter the start state: A

Enter the goal state: G

Path found:

Action: Move to B, State: B

Action: Move to D, State: D

Action: Move to E, State: E

Action: Move to G, State: G

**4.Implement AO\* Search algorithm**

import heapq

class Node:

def \_\_init\_\_(self, state, parent=None, action=None, cost=0, heuristic=0):

self.state = state # Current state in the search space

self.parent = parent # Parent node

self.action = action # Action that led to this node from the parent node

self.cost = cost # Cost to reach this node from the start node

self.heuristic = heuristic # Heuristic estimate of the cost to reach the goal

def \_\_lt\_\_(self, other):

return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def parse\_graph\_input():

graph = {}

num\_edges = int(input("Enter the number of edges: "))

for \_ in range(num\_edges):

u, v, cost = input("Enter an edge (format: u v cost): ").split()

cost = float(cost)

if u not in graph:

graph[u] = []

if v not in graph:

graph[v] = []

graph[u].append((v, cost))

return graph

def ao\_star\_search(start\_state, goal\_state, graph):

frontier = []

heapq.heappush(frontier, Node(start\_state, None, None, 0, heuristic(start\_state, goal\_state)))

explored = {}

while frontier:

current\_node = heapq.heappop(frontier)

current\_state = current\_node.state

if current\_state == goal\_state:

# Reconstruct the path from the goal node to the start node

path = []

while current\_node.parent is not None:

path.append((current\_node.action, current\_node.state))

current\_node = current\_node.parent

path.reverse()

return path

if current\_state not in explored or current\_node.cost < explored[current\_state]:

explored[current\_state] = current\_node.cost

for neighbor, step\_cost in graph.get(current\_state, []):

new\_cost = current\_node.cost + step\_cost

new\_node = Node(neighbor, current\_node, f"Move to {neighbor}", new\_cost, heuristic(neighbor, goal\_state))

heapq.heappush(frontier, new\_node)

return None # No path found

def heuristic(state, goal\_state):

# Simple heuristic function (e.g., straight-line distance)

heuristic\_values = {'A': 5, 'B': 3, 'C': 2, 'D': 1, 'E': 2, 'G': 0} # Custom heuristic values based on problem domain

return heuristic\_values.get(state, float('inf')) # Default to infinity if state not found

if \_\_name\_\_ == "\_\_main\_\_":

# Get user input to define the graph

print("Define the graph:")

graph = parse\_graph\_input()

start\_state = input("Enter the start state: ")

goal\_state = input("Enter the goal state: ")

# Perform AO\* search using the defined graph, start state, and goal state

path = ao\_star\_search(start\_state, goal\_state, graph)

# Print the resulting path found by AO\* search

if path:

print("Path found:")

for action, state in path:

print(f"Action: {action}, State: {state}")

else:

print("No path found.")

**out put**

Define the graph:

Enter the number of edges: 8

Enter an edge (format: u v cost): S A 3

Enter an edge (format: u v cost): S B 2

Enter an edge (format: u v cost): A C 4

Enter an edge (format: u v cost): A D 2

Enter an edge (format: u v cost): B E 3

Enter an edge (format: u v cost): C G 2

Enter an edge (format: u v cost): D G 5

Enter an edge (format: u v cost): E G 4

Enter the start state: S

Enter the goal state: G

Path found:

Action: Move to B, State: B

Action: Move to E, State: E

Action: Move to G, State: G

**5.Solve 8-Queens Problem with suitable assumptions**

def is\_safe(board, row, col):

""" Check if it's safe to place a queen at board[row][col] """

# Check column

for i in range(row):

if board[i][col] == 1:

return False

# Check upper diagonal on left side

i, j = row, col

while i >= 0 and j >= 0:

if board[i][j] == 1:

return False

i -= 1

j -= 1

# Check upper diagonal on right side

i, j = row, col

while i >= 0 and j < len(board):

if board[i][j] == 1:

return False

i -= 1

j += 1

return True

def solve\_queens(board, row):

""" Recursively solve the 8-Queens Problem using backtracking """

n = len(board)

# Base case: If all queens are placed, return True

if row >= n:

return True

for col in range(n):

if is\_safe(board, row, col):

board[row][col] = 1 # Place the queen

# Recur to place the rest of the queens

if solve\_queens(board, row + 1):

return True

# If placing queen at board[row][col] doesn't lead to a solution, backtrack

board[row][col] = 0 # Backtrack

return False

def print\_board(board):

""" Print the board configuration """

n = len(board)

for i in range(n):

for j in range(n):

print(board[i][j], end=" ")

print()

def solve\_8queens():

""" Solve the 8-Queens Problem and print the solution """

n = 8 # Size of the chessboard (8x8)

board = [[0] \* n for \_ in range(n)] # Initialize empty board

if solve\_queens(board, 0):

print("Solution found:")

print\_board(board)

else:

print("No solution exists.")

# Call the function to solve the 8-Queens Problem

solve\_8queens()

**out put**

Solution found:

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

**6.Implementation of TSP using heuristic approach**

import sys

def nearest\_neighbor\_tsp(distances):

num\_cities = len(distances)

# Start from the first city (arbitrary choice)

tour = [0] # Store the tour as a list of city indices

visited = set([0]) # Track visited cities

current\_city = 0

total\_distance = 0

while len(visited) < num\_cities:

nearest\_city = None

min\_distance = sys.maxsize

# Find the nearest unvisited city

for next\_city in range(num\_cities):

if next\_city not in visited and distances[current\_city][next\_city] < min\_distance:

nearest\_city = next\_city

min\_distance = distances[current\_city][next\_city]

# Move to the nearest city

tour.append(nearest\_city)

visited.add(nearest\_city)

total\_distance += min\_distance

current\_city = nearest\_city

# Complete the tour by returning to the starting city

tour.append(0)

total\_distance += distances[current\_city][0]

return tour, total\_distance

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

# Example distance matrix (symmetric, square matrix)

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

distances = [[ 0, 4, 8, 9, 12], [ 4, 0, 6, 8, 9], [ 8, 6, 0, 10, 11], [ 9, 8, 10, 0, 7], [12, 9, 11, 7, 0]]

# Run nearest neighbor TSP algorithm

tour, total\_distance = nearest\_neighbor\_tsp(distances)

# Print the tour and total distance

print("Nearest Neighbor TSP Tour:", tour)

print("Total Distance:", total\_distance)

**out put**

Nearest Neighbor TSP Tour: [0, 1, 2, 3, 4, 0]

Total Distance: 39

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

def forward\_chaining(rules, facts, goal):

inferred\_facts = set(facts)

new\_facts = True

while new\_facts:

new\_facts = False

for rule in rules:

condition, result = rule

if all(cond in inferred\_facts for cond in condition) and result not in inferred\_facts:

inferred\_facts.add(result)

new\_facts = True

if result == goal:

return True

return False

def backward\_chaining(rules, facts, goal):

def ask(query):

if query in facts:

return True

for rule in rules:

condition, result = rule

if result == query and all(ask(cond) for cond in condition):

return True

return False

return ask(goal)

# Define the rules and facts for the animal classification problem

rules = [

(['hair', 'live young'], 'mammal'),

(['feathers', 'fly'], 'bird')

]

facts = ['hair', 'live young']

goal = 'mammal'

# Use forward chaining to determine if a cat is classified as a mammal

is\_mammal = forward\_chaining(rules, facts, goal)

if is\_mammal:

print("The cat is classified as a mammal.")

else:

print("The cat is not classified as a mammal.")

facts = ['feathers', 'fly']

goal = 'bird'

# Use backward chaining to determine if a pigeon is classified as a bird

is\_bird = backward\_chaining(rules, facts, goal)

if is\_bird:

print("The pigeon is classified as a bird.")

else:

print("The pigeon is not classified as a bird.")

**out put**

Using forward chaining the cat is classified as a mammal.

Using backward chaining the pigeon is classified as a bird**.**

**8.Implement resolution principle on FOPL related problems**

def negate\_literal(literal):

""" Negate a literal by adding or removing the negation symbol '~' """

if literal.startswith('~'):

return literal[1:] # Remove negation

else:

return '~' + literal # Add negation

def resolve(clause1, clause2):

""" Resolve two clauses to derive a new clause """

new\_clause = []

resolved = False

# Copy literals from both clauses

for literal in clause1:

if negate\_literal(literal) in clause2:

resolved = True

else:

new\_clause.append(literal)

for literal in clause2:

if negate\_literal(literal) not in clause1:

new\_clause.append(literal)

if resolved:

return new\_clause

else:

return None # No resolution possible

def resolution(propositional\_kb, query):

""" Use resolution to prove or disprove a query using propositional logic """

kb = propositional\_kb[:]

kb.append(negate\_literal(query)) # Add negated query to knowledge base

while True:

new\_clauses = []

n = len(kb)

resolved\_pairs = set() # Track resolved pairs to avoid redundant resolutions

for i in range(n):

for j in range(i + 1, n):

clause1 = kb[i]

clause2 = kb[j]

if (clause1, clause2) not in resolved\_pairs:

resolved\_pairs.add((clause1, clause2))

resolvent = resolve(clause1, clause2)

if resolvent is None:

continue # No resolution possible for these clauses

if len(resolvent) == 0:

return True # Empty clause (contradiction), query is proved

if resolvent not in new\_clauses:

new\_clauses.append(resolvent)

if all(clause in kb for clause in new\_clauses):

return False # No new clauses added, query cannot be proven

kb.extend(new\_clauses) # Add new clauses to the knowledge base

# Example usage:

if \_name\_ == "\_main\_":

# Example propositional knowledge base (list of clauses)

propositional\_kb = [

['~P', 'Q'],

['P', '~Q', 'R'],

['~R', 'S']

]

# Example query to prove/disprove using resolution

query = 'S'

# Use resolution to prove or disprove the query

result = resolution(propositional\_kb, query)

if result:

print(f"The query '{query}' is PROVED.")

else:

print(f"The query '{query}' is DISPROVED.")

**out put**

The query 'S' is DISPROVED.

**9.Implement Tic-Tac-Toe game using Python.**

def print\_board(board):

""" Print the current state of the Tic-Tac-Toe board """

for row in board:

print(" | ".join(row))

print("-" \* 9)

def check\_winner(board, player):

""" Check if the specified player has won the game """

for row in board:

if all(cell == player for cell in row):

return True

for col in range(3):

if all(board[row][col] == player for row in range(3)):

return True

if all(board[i][i] == player for i in range(3)):

return True

if all(board[i][2-i] == player for i in range(3)):

return True

return False

def is\_full(board):

""" Check if the board is completely filled """

return all(cell != ' ' for row in board for cell in row)

def tic\_tac\_toe():

""" Main function to run the Tic-Tac-Toe game """

board = [[' ' for \_ in range(3)] for \_ in range(3)]

current\_player = 'X'

while True:

print\_board(board)

print(f"Player {current\_player}'s turn.")

row = int(input("Enter row (1-3): "))

col = int(input("Enter column (1-3): "))

row -= 1

col -= 1

if board[row][col] == ' ':

board[row][col] = current\_player

else:

print("Invalid move! Try again.")

continue

# Check if the current player has won

if check\_winner(board, current\_player):

print\_board(board)

print(f"Player {current\_player} wins!")

break

# Check if the board is full (tie game)

if is\_full(board):

print\_board(board)

print("It's a tie!")

break

# Switch to the other player

current\_player = 'O' if current\_player == 'X' else 'X'

if \_\_name\_\_ == "\_\_main\_\_":

tic\_tac\_toe()

**out put**

| |

---------

| |

---------

| |

---------

Player X's turn.

Enter row (1-3): 2

Enter column (1-3): 2

| |

---------

| X |

---------

| |

---------

Player O's turn.

Enter row (1-3): 1

Enter column (1-3): 1

O | |

---------

| X |

---------

| |

---------

Player X's turn.

Enter row (1-3): 1

Enter column (1-3): 2

O | X |

---------

| X |

---------

| |

---------

Player O's turn.

Enter row (1-3): 3

Enter column (1-3): 2

O | X |

---------

| X |

---------

| O |

---------

Player X's turn.

Enter row (1-3): 1

Enter column (1-3): 3

O | X | X

---------

| X |

---------

| O |

---------

Player O's turn.

Enter row (1-3): 3

Enter column (1-3): 3

O | X | X

---------

| X |

---------

| O | O

---------

Player X's turn.

Enter row (1-3): 3

Enter column (1-3): 1

O | X | X

---------

| X |

---------

X | O | O

---------

Player X wins!

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

import wikipedia# Import the Wikipedia library

def search\_and\_provide\_info(query):

try:

# Use the Wikipedia library to search for the query

results = wikipedia.search(query)

if not results:

return "No relevant information found."# Return if no results are found

# Select the first result and fetch its summary

page\_summary = wikipedia.summary(results[0], sentences=2)

return page\_summary

except wikipedia.exceptions.DisambiguationError as e:

# If there are multiple results, provide suggestions

return f"Multiple results found. Try to be more specific: {', '.join(e.options[:5])}"

except wikipedia.exceptions.PageError:

return "No information found for the given query."# Return if no page exists for the query

except Exception as e:

return f"An error occurred: {e}"# Return if any other exception occurs

def main():

while True:

# Take user input for the search query

query = input("Enter your search query (or 'quit' to exit): ")

if query.lower() == 'quit':

print("Exiting...")# Exit message

break# Break out of the loop

# Call the function to search and provide information

result = search\_and\_provide\_info(query)

# Print the result

print(result)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**

Enter your search query (or 'quit' to exit): Artificial Intelligence

Artificial intelligence (AI), in its broadest sense, is intelligence exhibited by machines, particularly computer systems. It is a field of research in computer science that develops and studies methods and software which enable machines to perceive their environment and uses learning and intelligence to take actions that maximize their chances of achieving defined goals

**11.Implement any Game and demonstrate the Game playing strategies**

import random # Import the random module

def rock\_paper\_scissors(): # Define the function for the game

choices = ['rock', 'paper', 'scissors'] # Define the choices that can be made

computer\_choice = random.choice(choices) # The computer makes a random choice

# The user is asked to make a choice

user\_choice = input("Enter your choice (rock, paper, scissors): ")

# If the user's choice is not valid, they are asked again

while user\_choice not in choices:

user\_choice = input("Invalid choice. Enter your choice (rock, paper, scissors): ")

# The choices of both the user and the computer are printed

print(f"\nUser choice: {user\_choice}")

print(f"Computer choice: {computer\_choice}")

# Determine the winner

if user\_choice == computer\_choice:

return "\nIt's a tie!"

if (user\_choice == 'rock' and computer\_choice == 'scissors') or \

(user\_choice == 'scissors' and computer\_choice == 'paper') or \

(user\_choice == 'paper' and computer\_choice == 'rock'):

return "\nUser wins!"

else:

return "\nComputer wins!"

print(rock\_paper\_scissors()) # Call the function to start the game

**Output**:

Enter your choice (rock, paper, scissors): rock

User choice: rock

Computer choice: rock

It's a tie!