VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Rushila V (1BM22CS226)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried outby Rushila V **(1BM22CS226),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Sneha P Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/Rushila226/AI-

Program 1
Tic Tac Toe

Algorithm:

Tic-Tac-Toe game

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

1) A function to define the board (3×3)
   def board-def
   print ("|".join row( )
   print ("-"*9)

2) We take three functions
   First function checks if all the elements in the row are same for i in range(3)
   if (board [i][0] == board [i][i] == board [i][2] != " ")
   return board [i][0]

   Second function checks if all the elements in the column are same
   if (board [0][i] == board [i][i] == board [2][i] != " ")
   return board [0][i]

   Third function checks if all the elements in the diagonal are same
   if (board[0][0] == board [1][1] == board [2][2] != " ") → return board[0][0]
   if (board [0][2] == board [1][1] == board[2][0] != " ")
   return board [0][2]

3) Next function checks if the spaces are full
   if (cell != " ")
   print ("All the cells are full. Try again")

4) Next is the main function
   for row in range(3) for col in range(3)
   row = int (input ("Enter the row: ")
   col = int (input ("Enter the col: ")
   ch = "X"
   if (board [row][col] == " ")
   board [row][col] = ch
   else
   print ("Get computer move")
   row, col = move (board)

5) In the final function move we generate a random choice for empty cells

6) def check winner (board)
   This function will return the winner of the game

Refer Algorithm

```
import random
def-print board(board)
  for row in board:
     print("|".join(row))
     print("-"*9)
def-check-winner(board):
  for i in range(3):
    if (board[i][0]==board[i][1]==board[i][2]!=" "):
      return board[i][0]
    if (board[0][i]==board[1][i]==board[2][i]!=" "):
      return board[0][i]
    if (board[0][0]==board[1][1]==board[2][2]!=" "):
      return board[0][0]
    if (board[0][2]==board[1][1]==board[2][0]!=" "):
      return board[0][2]
  return None
def-is full(board):
  return all(cell!=" " for row in board for cell in row)
def get-computer_move(board)
  empty-cells[(i,j) for i in range(3) for j in range(3)
  if board[i][j]==" ")
  return random.choice(empty-cells)
def tic-tac.toe():
  board=[[" " for-in range(3)] for-in range(3)]
  current-player:"X"
  computer-player:"O"
  while True:
    print-board(board)
    if current-player:"X":
      row=int(input("Player X enter the row (0-2): "))
      col=int(input("Player X enter the col (0-2): "))
```

```
else:
    print("Computer's turn")
    row, col = get_computer_move(board)
    print(f"Computer chooses row {row}, column {col}")
if board[row][col] == " ":
    board[row][col] = current_player
else:
    print("Cell is already taken! Try again")
    continue
winner = check_winner(board)
if winner:
    print_board(board)
    print(f"Player {winner} wins!")
    break
if is_full(board):
    print_board(board)
    print("It's a tie!")
    break
current_player = computer_player if current_player == "X"
 else "X"

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

24/9

Code:
```
import random

def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def check_winner(board):
    # Check rows, columns, and diagonals for a winner
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != " ":
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != " ":
            return board[0][i]

    if board[0][0] == board[1][1] == board[2][2] != " ":
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return board[0][2]

    return None

def is_full(board):
    return all(cell != " " for row in board for cell in row)

def get_computer_move(board):
    # Find the first empty cell for simplicity
    empty_cells = [(r, c) for r in range(3) for c in range(3) if board[r][c] == " "]
    return random.choice(empty_cells)  # Randomly pick one

def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"

    while True:
        print_board(board)
```

```
if current_player == "X": # Human
    player's turn
    row = int(input(f"Player {current_player}, enter the row (0-2): ")) col =
    int(input(f"Player {current_player}, enter the column (0-2): "))
else:
    # Computer's turn
    row, col = get_computer_move(board)
    print(f"Computer ({current_player}) chooses: row {row}, column {col}")

if board[row][col] == " ": board[row][col] =
    current_player
else:
    if current_player == "X":
        print("Cell is already taken! Try again.") continue

winner = check_winner(board) if winner:
    print_board(board) print(f"Player {winner}
    wins!") break

if is_full(board):
    print_board(board) print("It's
    a tie!") break

current_player = "O" if current_player == "X" else "X"

if _name_== "_main_":
tic_tac_toe()
```

```
Player X, enter the row (0-2): 0
Player X, enter the column (0-2): 2
  |   | X
---------
  |   |
---------
  |   |
---------
Computer (O) chooses: row 2, column 1
  |   | X
---------
  |   |
---------
  | O |
---------
Player X, enter the row (0-2): 1
Player X, enter the column (0-2): 2
  |   | X
---------
  |   | X
---------
  | O |
---------
Computer (O) chooses: row 0, column 1
  | O | X
---------
  |   | X
---------
  | O |
---------
```

```
Player X, enter the row (0-2): 2
Player X, enter the column (0-2): 2
  | O | X
---------
  |   | X
---------
  | O | X
---------
Player X wins!
```

Program 2
Vacuum cleaner problem

Algorithm:

Vacuum cleaner problem

```
def_init_(self)
  self.state {
  vacuum_pos = input ("Enter A or B")
  room.A : input("Enter clean or dirty")
  room.B : input ("Enter clean or dirty")

def move.left() {
  if (self.state(vacuum.pos) : : 'B')
     self.state (vacuum-pos) = : 'A'

def move.right() {
  if (self.state (vacuum.pos) : : 'A')
     self.state (vacuum.pos) : : 'B'

def suck() {
  if ( self.state (vacuum-pos) : : 'A')
  if(self.state (room.A) : : dirty)
     self.state (room.A) = "clean"

  else if (self.state (vacuum.pos) : : 'B')
     if (self.state (room.B) : : "dirty")
       self.state (room.B) : "clean"

def main()
  if (self.state (vacuum.pos) : : 'A')
    if (self.state (room.A) : : "dirty")
       suck()
    else
       move.right()
  else if (self.state (vacuum.pos) : : 'B')
    if (self.state (room.B) : : "dirty")
       suck()
    else
       move.left()
  else if (room.A : : "clean" as room.B : : "clean")
    break
```

```
class VacuumCleaner:
  def_init_(self):
    self.state = {
      "vacuum_pos": input("Enter the initial position of
      the vacuum cleaner (A or B): ").upper(),
      "room_A": input("Is Room A dirty or clean? ").lower(),
      "room_B": input("Is Room B dirty or clean? ").lower()

  }
def show_state(self):
  print(f"Vacuum Position: {self.state['vacuum_pos']},
  Room A: {self.state['room_A']},
  Room B: {self.state['room_B']}")

def_is clean(self):
  return self.state["room_A"]=="clean" and
   self.state("room_B")== "clean"

def move_right(self):
  if self.state("vacuum_pos"]=="A":
     self.state("vacuum_pos"]=="B"
     print("Moving to Room B")

def move_left(self):
  if self.state["vacuum_pos"]=="B":
     self.state["vacuum_pos"]=="A"
     print("moving to Room A")

def suck(self):
  if self.state["vacuum_pos"]=="A":
    if self.state["room_A"]=="dirty":
      self.state["room_A"]= "clean"
      print("Cleaning room B")
```

```
def run(self):
  while not self.is_clean():
    self.show-state()
    if self.state["vacuum-pos"]== "A":
      if self.state["room-A"]== "dirty":
        self.suck()
    else:
        self.move-right()
    elif self.state ["vacuum-pos"]== "B":
      if self.state ["room-B"] = = "dirty":
        self.suck()
    else:
        self.move-left()
  print ("Both rooms are clean now")
  self.show-state()
vacuum: VacuumCleaner()
vacuum.run()
```

Output:

Enter initial position of the vacuum cleaner (A or B): A
Is Room A dirty or clean? clean
Is Room B dirty or clean? dirty
Vacuum Position: A, Room A: clean, Room B: dirty
Moving to Room B
Vacuum Position: B Room A: clean, Room B: dirty
Cleaning Room B
Both rooms are clean now!
Vacuum position: B, Room A: clean. Room B: clean

1/10/24

Output for four rooms:

step 1

Vacuum is in room A, Room state: clean
Action: move down
moved down to Room C
Room states: {'A': 'clean', 'B': 'clean', 'C': 'Dirty', 'D': Dirty}

step 2

Vacuum is in room C, Room state: Dirty
Action: suck
sucking dirt in room C
Room states: {'A': 'clean', 'B': 'clean', 'C': 'clean', 'D': 'Dirty'}

step 3:

Vacuum is in room C; Room state: clean
Action: move right
moved right to room D
Room states: {'A': 'Clean', 'B': 'Clean', 'C': 'clean', 'D': 'Dirty'}

step 4:

Vacuum is in room D, Room state: Dirty
Action: suck
sucking dirt in room D
Room states: {'A': 'clean', 'B': 'clean', 'C': 'clean', 'D': clean}

Program

```
self.rooms = {
    'A': 'Clean',
    'B': 'Clean',
    'C': 'Dirty',
    'D': 'Dirty',

self.neighbors = {
    'A': {'right': 'B', 'down': 'C'},
    'B': {'left': 'A', 'down': 'D'},
    'C': {'up': 'A', 'right': 'D'},
    'D': {'up': 'B', 'left': 'C'}
```

Code:
```
def reflex_vacuum_agent(location, status):
    if status == 'Dirty':
        return 'Suck'
    elif location == 'A':
        return 'Right'
    elif location == 'B':
        return 'Left'

def main():
    print("Vacuum Cleaner Problem Simulation")

    environment = {}
    environment['A'] = input("Enter status for location A (Clean/Dirty): ").strip()
    environment['B'] = input("Enter status for location B (Clean/Dirty): ").strip()
    current_location = input("Enter starting location of the agent (A/B): ").strip()

    if environment['A'] not in ['Clean', 'Dirty'] or environment['B'] not in ['Clean', 'Dirty']:
        print("Invalid input! Status must be 'Clean' or 'Dirty'.")
        return

    if current_location not in ['A', 'B']:
        print("Invalid starting location! Must be 'A' or 'B'.")
        return

    while True:
        print(f"\nCurrent location: {current_location}")
        print(f"Status: {environment[current_location]}")

        action = reflex_vacuum_agent(current_location, environment[current_location])
        print(f"Action: {action}")

        if action == 'Suck':
            environment[current_location] = 'Clean'
            print(f"Cleaned location {current_location}")
        elif action == 'Right':
            current_location = 'B'
        elif action == 'Left':
```

```python
            current_location = 'A'

        print(f"Updated Environment: {environment}")

        if all(status == 'Clean' for status in environment.values()):
            print("\nBoth locations are clean. Stopping simulation.")
            break

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

```
Vacuum Cleaner Problem Simulation
Enter status for location A (Clean/Dirty): Dirty
Enter status for location B (Clean/Dirty): Dirty
Enter starting location of the agent (A/B): A

Current location: A
Status: Dirty
Action: Suck
Cleaned location A
Updated Environment: {'A': 'Clean', 'B': 'Dirty'}

Current location: A
Status: Clean
Action: Right
Updated Environment: {'A': 'Clean', 'B': 'Dirty'}

Current location: B
Status: Dirty
Action: Suck
Cleaned location B
Updated Environment: {'A': 'Clean', 'B': 'Clean'}

Both locations are clean. Stopping simulation.
```

Program 3

8 Puzzle game

Algorithm:

8 Puzzle Game

goal state:

{[1 2 3
4 5 6
7 8 0]}

moves: { down (1,0) left (0,1) up (-1,0) right (0,-1) }

def manhattan(state)
 for i in range(3)
  for j in range(3)
  if (state[i][j]) != 0) for of elem in goal stat we check the range according to 3x3 matrix
  goal_i, goal_j : divmod(state[i][j] - 1, 3)
  distance: goal_i + goal_j

def getneighbours
 for i in range(3)
  for j in range(3)
   pos_i = i + move[0]
   pos_j = i + move[1]

This function gets the neighbouring element once the puzzle is shuffled

this above logic gets the puzzles arranged using manhattan distance where we calculate the horizontal and vertical distance of the element in the goal state from initial state

Next in the dfs function we take two variables visited and unvisited all the elements that have been checked and placed in correct order are assigned in visited list and all the elements that are not checked are placed in unvisited list we do this so that the same element is not checked in the new state again and again

1	2	3
4	5	6
7	8	0

3	1	8
7	6	4
0	2	5

```
curr_state = { }
goal_state = { }
stack.push(curr_state)
moves = 0

if (curr_state == goal_state)
{
    left = (0,1)        row
                        column
    up = (-1,0)
    right (0,-1)
    down = (1,0)
}
print (moves)
```

```
from collections import deque
GOAL_STATE = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]
MOVES = [
    (-1, 0).
    (1, 0),
    (0, -1),
    (0, 1)
]
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i, goal_j = divmod(state[i][j] - 1, 3)
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance
def _is_goal_state(state):
    return state == GOAL_STATE

def get_neighbours(state):
    neighbors = []
    for i in range(3):
        for j in range(3):
            if (state[i][j] == 0):
                for move in MOVES:
                    ni, nj = i + move[0], j + move[1]
                    if 0 <= ni < 3 and 0 <= nj < 3:
                        newstate = [row[:] for row in state]
                        new_state[i][j], newstate[ni][nj] = new_state[ni][nj], new_state[i][j]
                        neighbors.append(newstate)
    return neighbors
```

```
def dfs(state):
  queue : deque([(state, [state])])
  visited = set()
  while queue:
    cs, p = queue.popleft()
    if is_goal_state(cs):
      return p
    if tuple(map(tuple, cs)) in visited:
      continue
    visited.add(tuple(map(tuple, cs)))
      for neighbor in get_neighbors(cs):
        queue.append((neighbor, path p + [neighbor]))
  return None
initial_state = [
   [4,1,3],
   [7,2,6],
   [5,8,0]
 ]
path = dfs(initial_state)
if p:
   print("solution found")
   for state in p:
      for row in state:
         print(row)
      print()
else:
   print("No solution found")
```

Output

Solution found:

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

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

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

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

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

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

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

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

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

4	1	3
7	2	6
5	8	0

1	2	3
4	5	6
7	8	0

1 + 1 + 0 + 1 + 3 + 0 + 0 + 2 : 8

total moves : 8

8/10/24

Code:

```
from collections import deque

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

MOVES = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i = (state[i][j] - 1) // 3  # Calculate the goal row
                goal_j = (state[i][j] - 1) % 3   # Calculate the goal column
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance

def is_goal_state(state):
    return state == GOAL_STATE

def get_neighbors(state):
    neighbors = []
    zero_pos = None

    # Find the position of the empty space (0)
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                zero_pos = (i, j)
                break
        if zero_pos:
            break

    for move in MOVES:
```

```
        new_i, new_j = zero_pos[0] + move[0], zero_pos[1] + move[1] if 0 <=
        new_i < 3 and 0 <= new_j < 3:
            new_state = [row[:] for row in state] new_state[zero_pos[0]][zero_pos[1]],
            new_state[new_i][new_j] =
new_state[new_i][new_j], new_state[zero_pos[0]][zero_pos[1]]
            neighbors.append(new_state)

    return neighbors def

dfs(initial_state):
    queue = deque([(initial_state, [initial_state])])
    visited = {tuple(map(tuple, initial_state))}

    while queue:
        current_state, path = queue.popleft() if
        is_goal_state(current_state):
            return path
        for neighbor in get_neighbors(current_state):
            neighbor_tuple = tuple(map(tuple, neighbor)) if
            neighbor_tuple not in visited:
                visited.add(neighbor_tuple) queue.append((neighbor,
                path + [neighbor]))

    return None

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

path = dfs(initial_state) if path:
    print("Solution found:") for state
    in path:
        for row in state: print(row)
        print()
else:
    print("No solution found.")
```

```
Solution found:
[4, 1, 3]
[7, 2, 6]
[5, 8, 0]

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

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

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

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

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

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

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

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

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

Program 4

Implement Iterative deepening search algorithm

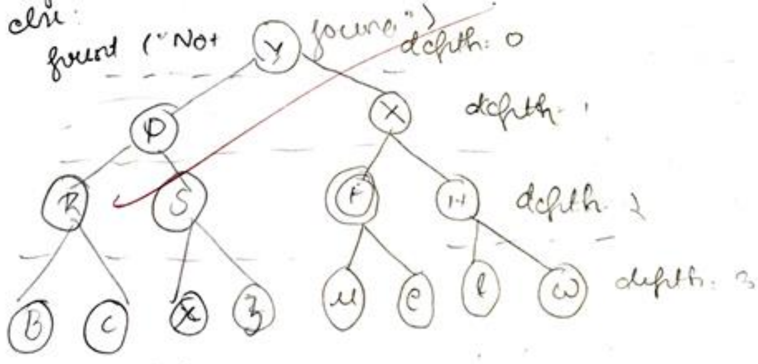
Algorithm:

Iterative deepening search 15/10/24

```
def ids (start, goal, graph)
if depth :: 0 :
    if node :: goal
      return node

elif depth >0:
  result ids (child, goal, depth - 1)
if result is not None
   return [node] + result

depth : 0
while True
  result : ids (start, goal, depth)
if result is not None
    return result
  depth : depth + 1
def main ( )
  noe : input ("Enter the number of edges")
start node = input ("Enter start node")
 goal_node = input ("Enter goal node")
 if path :
   print ("Path found")
else :
   print ("Not found")
```



level 0 : Y

level 1 : Y → P → X

level 2 : Y → P → R → S → X → F

Iterative deepening search

```
def ids(graph, start, goal):
  def dls(node, goal, depth):
    if depth == 0:
      if node == goal:
        return [node]
    else:
      return None

    elif depth>0:
      for child in graph.get(node,[]):
        result = dls(child, goal, depth-1)
        if result is not None:
          return [node] + result
    return None

  depth = 0
  while true:
    result = dls(start, goal, depth)
    if result is not None:
      return Result
    depth += 1

def main():
  graph = {}
  noe = int(input("Enter number of edges"))
  print("Enter each edge in the format node1 node2")
  for _ in range(noe):
    node1, node2 = input().split()
  if node1 in graph:
    graph[node1].append(node2)
  else:
    graph[node1] = [node2]
  if node2 in graph:
    graph[node2].append(node1)
```

```
else:
    graph[node2] = [node1]
return graph
def main():
    graph = main()
    s_node : input ("Enter starting node: ")
    g_node: input ("Enter goal node: ")
    path = iids (graph, s_node, g_node)
    if path:
        print ("Path found")
    else
        print ("No path found")

if _name_ = = "_main_":
    main()
```

O/P:

```
Enter the number of edges: 14
Enter each edge
YP   RC
YX   SX
PR   SZ
PS   FU
XF   FC
XH   HL
RB   HW
Enter starting node: Y
Enter goal node: F
Path found: Y→X→F
```

Code:

```
class Node:
    def __init__(self, state, parent=None, depth=0):
        self.state = state
        self.parent = parent
        self.depth = depth

    def path(self):
        # Construct path from root to the current node
        node, path = self, []
        while node:
            path.append(node.state)
            node = node.parent
        return path[::-1]

    def __repr__(self):
        return f"Node({self.state}, depth={self.depth})"


def depth_limited_search(start, goal, depth_limit, graph):

    stack = [Node(start)]
    while stack:
        current = stack.pop()
        if current.state == goal:
            return current
        if current.depth < depth_limit:
            for neighbor in graph.get(current.state, []):
                stack.append(Node(neighbor, parent=current, depth=current.depth + 1))
    return None


def iterative_deepening_search(start, goal, graph):

    depth = 0
    while True:
        print(f"Trying depth limit: {depth}")
        result = depth_limited_search(start, goal, depth, graph)
```

```
        if result:  return
            result
        depth += 1


if __name___== "__main__":   print("Iterative
    Deepening Search (IDS)")

    graph = {}
    num_edges = int(input("Enter the number of edges: "))
    print("Enter each edge (e.g., A B for an edge from A to B):") for
    _ in range(num_edges):
        u, v = input().split() if
        u not in graph:
            graph[u] = []
        graph[u].append(v)

    start_state = input("Enter the starting node: ").strip()
    goal_state = input("Enter the goal node: ").strip()

    print("\nStarting Iterative Deepening Search...")
    solution = iterative_deepening_search(start_state, goal_state, graph)

    if solution:
        print(f"Goal found! Path: {' -> '.join(solution.path())}") else:
        print("Goal not found.")
```

```
Iterative Deepening Search (IDS)
Enter the number of edges: 6
Enter each edge (e.g., A B for an edge from A to B):
A B
A C
B D
B E
C F
C G
Enter the starting node: A
Enter the goal node: F

Starting Iterative Deepening Search...
Trying depth limit: 0
Trying depth limit: 1
Trying depth limit: 2
Goal found! Path: A -> C -> F
```

Program 5

8 Puzzle game using A* search algorithn

Algorithm:

8 Puzzle game

Initial state

1	2	3
8		4
7	6	5

goal state

2	8	1
	4	3
7	6	5

A* algorithm $f(N) = H(N)+G(N)$

def H.n(state, target)
 return sum (x!=y)

The function fn is used to decide which state to explore next. it adds the moves taken and remaining moves to determine cost

def possible moves (state, lvl, visited_states).

directions = {'d': down, 'u': up, 'l': left, 'r': right}

if b<=5 directions.append('d')

if b>=3 directions append('u')

0,3,6

if b%3>0 directions.append('l')

if b%3<2 directions.append('r')

2,5,8

0 1 2
3 4 5
6 7 8

if temp not in visited_states:
 pos_moves.append([temp, lvl+1])

It checks if the new state hasn't been visited yet if not it adds to list of

function to generate a new state based on a move position. In this we create a copy of the current st ath mo to avoid modifying the original

temp[b], temp[b±n]: swap empty tile with its neighbor

for i in range(3)
 for j in range(3)
 display the state in 3x3 format

def astar(src, target):
 arr = [[src, 0]] // list of states to explor
 visited_states = []
 iterations = 0

display the current state if the current state matches the target state return the no of iteration

 display_state (current[0])
 if current [0] == target
 return iterations

In the main function add visited states to list the user should give initial state & goal state & call a function

Deroceed

A star

```
def H_n(state, target):
  return sum (x!= y for x, y in zip((state, target))

def F_n (sl , target):
  state, level : sl
  return H.n(state, target) + lvl

def possible_moves ( sl, visited_states):
 state, lvl : sl
 b : state.index(0)
 directions: []
 pos_moves: []
  if b<=5:
    directions.append('d')
  if b>=3:
    directions.append('u')
  if b%3>0:
    directions.append('l')
  if b%3<2:
    directions.append('r')
for move in directions:
  temp: gen(state, move, b)
  if temp not in visited_states:
    pos_moves.append([temp, lvl+1])

def gen (state, move, b):
  temp: state.copy()
  if move == 'l' : temp[b], temp[b-1] = temp[b-1], temp[b]
  if move == 'r' : temp[b], temp[b+1] = temp[b+1], temp[b]
  if move == 'u' : temp[b], temp[b-3] = temp[b-3], temp[b]
  if move == 'd' : temp[b], temp[b+3] = temp[b+3], temp[b]

return temp
```

```
def - display state (state):
  print ("Current state ")
for i in range (0,9,3):
   print (state [i: i+3])

print ()

def astar (src, target):
     arr : [[src,0) ]
     visited : [ ]
     i :0

while arr:
   i + = 1
   current = min (arr, key = lambda x: f_n (x,target))
   arr .remove (current )
   display_ state (current [0] )
   if current (0] :: target:
     return iterations

  visited_ states. append (current [0] )
  arr. extend ( possible _moves (current, visited_states))
 return "Notfound'
src : [1, 2 ,3, 8,0,4 ,7 ,6, 5 )
target: [2 ,8, 1 ,0, 4 ,3,7,6,5 ]
print ( astar (src, target ))
```

Output

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Code:

```
from heapq import heappop, heappush
class PuzzleNode:
    def __init__(self, state, parent=None, move=None, depth=0, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.depth = depth  # g(n): number of moves from start
        self.cost = cost  # f(n) = g(n) + h(n)

    def __lt__(self, other):
        return self.cost < other.cost

    def path(self):
        moves = []
        node = self
        while node.parent is not None:
            moves.append(node.move)
            node = node.parent
        return moves[::-1]


def manhattan_distance(state, goal):
    distance = 0
    for i, value in enumerate(state):
        if value == 0:
            continue
        goal_index = goal.index(value)
        distance += abs(i // 3 - goal_index // 3) + abs(i % 3 - goal_index % 3)
    return distance

def get_neighbors(state):
    neighbors = []
    zero_index = state.index(0)
    row, col = divmod(zero_index, 3)
    moves = {
        "Up": (-1, 0),
        "Down": (1, 0),
```

```
        "Left": (0, -1),
        "Right": (0, 1),
    }

    for move, (dr, dc) in moves.items():
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = state[:]
            new_state[zero_index], new_state[new_index] = new_state[new_index],
new_state[zero_index]
            neighbors.append((new_state, move))
    return neighbors

def a_star(start, goal):

    open_set = []
    closed_set = set()

    start_node = PuzzleNode(start, cost=manhattan_distance(start, goal))
    heappush(open_set, start_node)

    while open_set:
        current = heappop(open_set)

        if current.state == goal:
            return current

        closed_set.add(tuple(current.state))

        for neighbor_state, move in get_neighbors(current.state):
            if tuple(neighbor_state) in closed_set:
                continue

            depth = current.depth + 1
            cost = depth + manhattan_distance(neighbor_state, goal)
            neighbor_node = PuzzleNode(
                neighbor_state, parent=current, move=move, depth=depth, cost=cost
            )
            heappush(open_set, neighbor_node)
```

```
    return None

if __name___== "__main__":
    print("8-Puzzle Solver using A* Algorithm")
    print("Enter the start state (row-wise, 0 for the blank tile):") start
    = [int(x) for x in input().split()]
    print("Enter the goal state (row-wise, 0 for the blank tile):") goal
    = [int(x) for x in input().split()]

    if sorted(start) != sorted(goal):
        print("Invalid input! The start and goal states must contain the same elements.") else:
        solution = a_star(start, goal)

        if solution:
            print("\nSolution found!")
            print(f"Moves: {' -> '.join(solution.path())}")
            print(f"Number of moves: {len(solution.path())}")
        else:
            print("\nNo solution exists.")
```

```
8-Puzzle Solver using A* Algorithm
Enter the start state (row-wise, 0 for the blank tile):
1 2 3 4 5 0 6 7 8
Enter the goal state (row-wise, 0 for the blank tile):
1 2 3 4 5 6 7 8 0

Solution found!
Moves: Down -> Left -> Left -> Up -> Right -> Down -> Right -> Up ->
    Left -> Left -> Down -> Right -> Right
Number of moves: 13
```

Program 6

Simulated Annealing algorithm

Algorithm:

```
import math                              → x value is 10
import random                            f(10) = (10-3)^2 = (7)^2 = 49
def objective function(x)
    return (x-3)**2

def simulated annealing (objective function, is, itemp,
cooling rate, st, iterations)
    current_sol = is  // 10
    current_val = objectivefunction(x, current_sol) // 49

    best_sol = current_sol
    best_val = current_val

    temp = itemp  // 1000
    iterations = 0  0 < 10
while iterations < max iteration and temp > stopping temp
    then generate a new value by taking a random
    value and adding to current_sol
ns = current_sol + random.uniform(-1, 1)
             10 + 0.5 = 10.5
 dv = new_val - current_val
           10.5 - 10 = 0.5
 if dv < 0
    If dv is negative it means the new solution has
    a lower value which makes it better for
    minimization
 current_sol = new_sol
 current_val = new_val

else if the new solution is worse don't accept it

Next check if the solution found is the best one
  if current_val < best_val
     best_sol = current_sol
     best_val = current_val

Increment the iterations
values: is = 10                        After every iteration
     max iteration = 10                  decrease the temp
     stopping_temp = 1e-8                by 5%
     itemp = 1000                        temp = temp *
     cooling rate = 0.95                 cooling rate
```

```
import math
import random
def objective_function (x):
    return (x-3)** 2
def simulated_annealing (objective_function, initial_solution,
initial_temperature, cooling_rate, stopping_temperature,
max_iterations):
  current_solution = initial_solution
  current_value = objective_function (current_solution)
  best_solution = current_solution
  best_value = current_value

  temperature = initial_temperature
  iteration = 0

  while temperature > stopping_temperature and
  iteration < max_iterations:
    new_solution = current_solution + random.uniform (-1,1)
    new_value = objective_function (new_solution)
    delta_value = new_value - current_value
    if delta_value<0:
       current_solution = new_solution
       current_value = new_value
    else:
      probability = math.exp(- delta_value/temperature)
      if random.random()< probability:
           current_solution = new_solution
           current_value = new_value

    if current_value < best_value:
       best_solution = current_solution
       best_value = current_value

    temperature = temperature * cooling rate
    iteration += 1
    print (f"iteration: {iteration}, Temperature: {temperature:4f}
    Current solution: {current_solution:4f}.
    Best solution: {best_solution:4f}
```

return best_solution, best_value

initial_solution: 10
initial_temperature: 1000
cooling_rate: 0.95
stopping temperature: 1e-8
max_iterations: 10

best_solution, best_value = simulated annealing (objective_function, initial_solution, initial_temperature, cooling_rate, stopping_temperature, max_iterations)
print(f "best solution:{4f}, f(x) : {best_value : 4f}")

Output:

iteration : 1 Temperature : 950.0000, current soln : 9.4775
iteration : 2 Temperature : 902.0000 current soln : 9.5096
iteration : 3 Temperature : 857.3750 current soln : 9.6366
iteration : 4 Temperature : 814.5062 current soln : 10.4510
iteration : 5 Temperature : 773.7809 current soln : 10.1823
iteration : 6 Temperature : 735.0918 current soln : 10.1549
iteration : 7 Temperature : 698.337 current soln : 10.6004
iteration : 8 Temperature : 663.4204 current soln : 10.993
iteration : 9 Temperature : 630.2494 current soln : 10.8792
iteration : 10 Temperature : 598.7369 current soln : 11.7439
Best soln : 9.475315
Best soln : 9.475315
Best soln : 9.475315
Best soln : 9.475315
Best soln : 9.475315
Best soln : 9.475315
Best soln : 9.475315
Best soln : 9.475315
Best soln : 9.475315
Best soln : 9.475315

8 22/10/24

Code:

```
import math import random

def objective_function(x):
    return (x - 3) ** 2

def simulated_annealing(objective_function, initial_solution, initial_temperature, cooling_rate,
stopping_temperature, max_iterations):

    current_solution = initial_solution
    current_value = objective_function(current_solution)

    best_solution = current_solution
    best_value = current_value

    temperature = initial_temperature
    iteration = 0

    while temperature > stopping_temperature and iteration < max_iterations:
        new_solution = current_solution + random.uniform(-1, 1)  # Take a small random step from
the current solution
        new_value = objective_function(new_solution)

        delta_value = new_value - current_value

        if delta_value < 0:
            current_solution = new_solution
            current_value = new_value
        else:
            probability = math.exp(-delta_value / temperature)
            if random.random() < probability:
                current_solution = new_solution
                current_value = new_value

        if current_value < best_value:
            best_solution = current_solution
            best_value = current_value
```

```
        temperature *= cooling_rate
        iteration += 1

        print(f"Iteration: {iteration}, Temperature: {temperature:.4f}, Current Solution:
{current_solution:.4f}, Best Solution: {best_solution:.4f}") return

    best_solution, best_value

initial_solution = 10  # Starting point for the algorithm
initial_temperature = 1000  # High starting temperature
cooling_rate = 0.95  # Rate at which the temperature is decreased (reduce by 5% each iteration)
stopping_temperature = 1e-8  # Algorithm stops when the temperature is very low max_iterations =
5  # Limit the number of iterations to prevent infinite loops

best_solution, best_value = simulated_annealing(objective_function, initial_solution,
initial_temperature, cooling_rate, stopping_temperature, max_iterations)

print(f"Best solution found: x = {best_solution:.4f}, f(x) = {best_value:.4f}")
```

```
Iteration: 1, Temperature: 950.0000, Current Solution: 9.5596, Best
    Solution: 9.5596
Iteration: 2, Temperature: 902.5000, Current Solution: 9.0908, Best
    Solution: 9.0908
Iteration: 3, Temperature: 857.3750, Current Solution: 8.7255, Best
    Solution: 8.7255
Iteration: 4, Temperature: 814.5062, Current Solution: 8.9231, Best
    Solution: 8.7255
Iteration: 5, Temperature: 773.7809, Current Solution: 9.7791, Best
    Solution: 8.7255
Best solution found: x = 8.7255, f(x) = 32.7809
```

Program 7

8 queens using A star algrithm:

Algorithm

A star algorithm

step1: create a 8x8 chessboard to initialise state

(b) set up an open set to explore different configurations

(c) set up a visited state to display all different configuration

step 2: calculate the number of attacking pairs that the queens should not be in the same row same column or same diagonal

if state[i] == state[j] or abs(state[i]) state[j] == j - 1 || diagonal

same row column attacks = 1

then we increment the variable attacks to determine attacking pair

open_set = []

step 3: Assign initial state to open set for the first iteration. If the node is not visited then first push it to the open set; the open set will push the node to heap q (priority queue)

heapq.heap-push(open_set, Node(new_state, g, h))

step 4: we calculate total estimated cost, f = g+h where g is the cost to reach the current state h is the no. of attacks to reach goal state

step 5: In the main loop remove the node with the lowest cost

step 6: we have to determine the next row to place the queen

step 7: this happens in the main loop after doing this we generate new state

update g and calculate h

A star algorithm

```
import heapq
class Node:
  def _init_ (self, state, g, h):
      self.state = state
      self.g = g
      self.h = h
      self.f = g+h
  def _lt_(self, other):
      return self.f < other.f
 def heuristic (state):
   attacks = 0
   for i in range(len(state)):
     for j in range (i+1, len(state)):
       if state(i) == state(j) or abs (state(i)-sta
         == j-i:
          attacks += 1
   return attacks

 def a-star-8queens():
    initial-state = tuple ([-1] * 8)
    open-set = [ ]
    heapq.heappush(open-set, Node(initial-state,
    heuristic (initial-state)))
    visited = set( )
    while open-set:
      current_node = heapq.heappop(open.set)
      current-state = current_node.state

if current-node.h == 0 and -1 not in current-stat:
   return current.state
```

```
if current_state in visited :
    continue
visited.add(current_state)
next_row = current_state.index(-1) if -1 in
current_state else len(current_state)
if next_row < 8:
    for col in range(8):
        new_state = list(current_state)
        new_state[next_row] = col
        new_state = tuple(new_state)
        if new_state not in visited:
            g = current_node.g + 1
            h = heuristic(new_state)
        heapq.heappush(open_list, Node(new_state, g, h))

return None

def display(state):
    for row in range(8):
    line = " "
      for col in range(8):
        if state[row] == col:
          line += "Q"
        else :
            line += " ."
    print(line)
  print()
solution = a_star_8_queens()
if solution:
    print("A*solution:")
    display(solution)
else:
    print("No solution found")
```

output

A* solution

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

Code:

```
import heapq

def is_valid(board, row, col):
    for c in range(col):
        r = board[c]
        if r == row or abs(r - row) == abs(c - col):  # Same row or diagonal
            return False
    return True

def heuristic(board, n):
    conflicts = 0
    for col1 in range(n):
        row1 = board[col1]
        if row1 == -1:
            continue
        for col2 in range(col1 + 1, n):
            row2 = board[col2]
            if row2 == -1:
                continue
            if row1 == row2 or abs(row1 - row2) == abs(col1 - col2):  # Row or diagonal conflict
                conflicts += 1
    return conflicts

def a_star_8_queens(n=8):
    initial_state = [-1] * n  # Empty board
    pq = []
    heapq.heappush(pq, (0, 0, initial_state))  # (f, g, board)

    while pq:
        f, g, board = heapq.heappop(pq)

        if g == n:  # Goal state
            return board

        col = g  # Place the next queen in the current column
```

```
        for row in range(n):
            if is_valid(board, row, col):
                new_board = board[:]
                new_board[col] = row
                h = heuristic(new_board, n)
                heapq.heappush(pq, (g + 1 + h, g + 1, new_board))

    return None  # No solution

# Solve the 8-Queens problem
solution = a_star_8_queens()
if solution:
    for col, row in enumerate(solution):
        print(f"Queen at column {col + 1}, row {row + 1}")
else:
  print("No solution found.")
```

```
Queen at column 1, row 1
Queen at column 2, row 5
Queen at column 3, row 8
Queen at column 4, row 6
Queen at column 5, row 3
Queen at column 6, row 7
Queen at column 7, row 2
Queen at column 8, row 4
```

Program 8
8 queens using hill climbing algorithm

Algorithm

Hill climbing algorithm

step 1: Place 8 queens randomly

step 2: Find the attacking pairs such that no queens are placed in the same row, same column or same diagonal

step 3: Place 8 queens randomly on the chessboard

state = [random.randint(0,7) for in range(8)]

current_attack = calculate_attack(state)

step 4: In the previous state calculate attacks choose one with few attacking pairs

step 5: if next_attack >= current_attack
    break

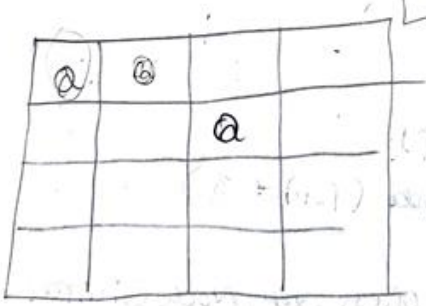
    update the current state

    state = next_state

    current_attack = new_attack

step 6: display board

$\boxed{f = g + h}$



Hill climbing algorithm

```
import random
def calculate-attacks (state):
  attacks = 0
  for i in range (len (state)):
    for j in range (i+1, len( state)):
      if state[i] == state(j) or abs (state[i]- state[j] == j-i
        attacks += 1
  return attacks

def hill ():
 state = [random. randint (0, 7) for _ in range (8))
  ca = calculate- attack(state)
  for _ in range (100):
     n = []
     for r in range(8):
       for c in range (8):
        if state[row] != col:
           new state [:]
           n[r] = c
           n. append(n )
  next- state = min (n, key = calculate_ attacks)
  next- attacks = calculate - attacks(next. state)
  if next-attacks >= current-attacks :
     break
   state = next - state
   current - attacks = next_attack
  return state, current-attack
def display (state):
  for r in range (8):
  line = " "
  for c in range (8):
    if state[r] == c:
      line += "Q"
   else:
```

```
    line += "."
print(line)
print()

best_solution = None
best_attacks = float('inf')
attempts = 100
for _ in range(attempts):
    s, a = hill();
    if attacks < best_attacks:
        best_solution = solution
        best_attacks = attacks
    if best_attacks == 0:
        break

if best_solution:
    print({best_attacks})
    display(best_solution)
else:
    print("No solution found")
```

Output:

Best solution found

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

29/10/24

Code:

```
import random

def calculate_heuristic(board):
    """Calculate the heuristic: number of pairs of queens attacking each other."""
    conflicts = 0
    n = len(board)
    for col1 in range(n):
        for col2 in range(col1 + 1, n):
            if board[col1] == board[col2] or abs(board[col1] - board[col2]) == abs(col1 - col2):  # Row or diagonal
conflict
                conflicts += 1
    return conflicts

def get_neighbors(board):
    """Generate all possible neighbors by moving each queen to a different row in its column."""
    n = len(board)
    neighbors = []
    for col in range(n):
        for row in range(n):
            if board[col] != row:
                new_board = board[:]
                new_board[col] = row
                neighbors.append(new_board)
    return neighbors

def hill_climbing(n=8):
    """Solve the N-Queens problem using Hill Climbing."""
    # Start with a random initial state
    current_board = [random.randint(0, n - 1) for _ in range(n)]
    current_h = calculate_heuristic(current_board)

    while True:
        neighbors = get_neighbors(current_board)
        neighbor_hs = [(calculate_heuristic(neighbor), neighbor) for neighbor in neighbors]
        min_h, best_neighbor = min(neighbor_hs, key=lambda x: x[0])

        if min_h >= current_h:  # Local minimum or no improvement
            break

        # Move to the neighbor with the lowest heuristic
        current_board = best_neighbor
        current_h = min_h

        if current_h == 0:  # Solution found
            break

    return current_board, current_h

# Solve the 8-Queens problem
solution, heuristic = hill_climbing()
```

```
if heuristic == 0:
    print("Solution found!")
    print("Board configuration (column: row):", solution)
else:
    print("No solution found. Stuck at local minimum.")
    print("Board configuration (column: row):", solution)
```

```
No solution found. Stuck at local minimum.
Board configuration (column: row): [2, 7, 5, 3, 0, 6, 4, 1]
```

Program 9

Create a knowledge base using propositional logic and show that the given query entails the knowledge base ornot.

Algorithm:

Lab-7

Knowledge Base :-

1) Alice is the mother of Bob
2) Bob is the father of Charlie
3) A father is a parent
4) A mother is a parent
5) All parents have children
6) If someone is a parent their children are siblings
7) Alice is married to David

Hypothesis :

- "Charlie is a sibling of Bob"

Entailment Process

From 1 and 2 :

Alice is the mother of Bob and Bob is the father of Charlie

From 3 and 4 :

father and mother are considered as a parent

From 5 :

All parents have children Bob is a child of Alice and charlie is a child of Bob.

From 6 & 7 :

since Bob is a parent of the Charlie & potentially Alice due to her marriage to David

Conclusion:

The hypothesis is true

8
19/11/24

Code:
```
# Define relationships as functions
def is_parent(person):
    return person in ["Alice", "Bob"]

def is_child(person):
    return person == "Charlie"

def is_sibling(person1, person2):
    # Siblings share at least one parent
    return person1 != person2 and all(is_parent(parent) for parent in ["Alice", "Bob"])

# Knowledge Base
def knowledge_base():
    rules = {
        "Alice is the mother of Bob": is_parent("Alice") and is_parent("Bob"),
        "Bob is the father of Charlie": is_parent("Bob") and is_child("Charlie"),
        "All parents have children": all(is_child("Charlie") for _ in ["Alice", "Bob"]),
    }
    return rules

# Hypothesis
def hypothesis():
    return is_sibling("Bob", "Charlie")

# Validate the entailment
def validate_hypothesis():
    kb = knowledge_base()
    if all(kb.values()) and hypothesis():
        print("The hypothesis is true based on the knowledge base.")
    else:
        print("The hypothesis is not necessarily true.")

# Run the validation
validate_hypothesis()
```

```
The hypothesis is true based on the knowledge base.
```

Program 10
First Order Logic

Algorithm:

Lab-8

Problem :-
(1) All birds can fly
(2) Tweety is a bird
(3) Tweety can fly
Prove: Tweety can fly

x: Bird

Proof:
1) Give $\forall x(B(x) \rightarrow F(x))$
(Premise: All birds can fly)

2) Given: B(Tweety)
(Premise: Tweety a bird)

3) Universal Instantiation:
From $\forall x(B(x) \rightarrow F(x))$
we conclude B(Tweety) → F(Tweety)

4) Modus Ponens: From B(Tweety) (of step 2) and
B(Tweety) → F(Tweety) we conclude F(Tweety)

Conclusion
We have proved that Tweety can fly

Output:
Tweety can fly

19/11/24

Consider the following problem:
As per the law, it is a crime for an American to sell weapons to hostile nations. Country A an enemy of America has some missiles and all the missiles of were sold to it by Robert, who is an American citizen.
Prove that "Robert is criminal"

Proof:

• American(p) $\wedge$ weapon(q) $\wedge$ sells(p, q, r) $\wedge$ Hostile $\Rightarrow$ Criminal

$p \rightarrow$ American
$q \rightarrow$ weapon
$r \rightarrow$ Hostile
$p \rightarrow$ Criminal (p)

• Country A has missiles. This can be expressed as:
$(\exists x\ (\text{Owns}(A, x) \wedge \text{missile}(x)))$
Country A owns an object x and that x is a missile

• Instantiating with a specific missile (T1):
$\text{Owns}(A, T1) \wedge \text{missile}(T1)$
We are picking a specific missile called T1 and replacing x with it

• $\forall x\ (\text{missile}(x) \wedge \text{Owns}(A, x)) \Rightarrow \text{Sells}(\text{Robert}, x, A)$
$\text{Sells}(\text{Robert}, T1, A)$ Robert sold missile T1 to country A
For every missile country owns Robert sold all the missiles to country A

• Missiles are weapons
$\text{missiles}(x) \Rightarrow \text{Weapons}(x)$

• Enemy of america which is country A is a hostile nation
$\forall x\ (\text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x))$

• Robert is an American
American (Robert)

Applying the law

American (Robert)

Weapon (T1)

Robert sold missile T1 to country A:

Sells (Robert, T1, A)

Hostile (A)

∴ American(p) ∧ Weapon (q) ∧ Sells (p, q, r) ∧ Hostile (r) ⇒ Criminal (p)

p = Robert

q = T1

r = A

American(Robert) ∧ Weapon (T1) ∧ Sells (Robert, T1, A) ∧ Hostile (A) ⇒ Criminal (Robert)

Proved

o/p:

Robert is a criminal

Code:

```
# Define predicates
def is_bird(x):
    # x is a bird
    return x == "Tweety"

def can_fly(x):
    # Birds can fly
    return is_bird(x)

# Proof process
def proof_tweety_can_fly():
    # Step 1: All birds can fly
    rule_all_birds_fly = lambda x: is_bird(x) and can_fly(x)

    # Step 2: Tweety is a bird
    fact_tweety_is_bird = is_bird("Tweety")

    # Step 3: Universal instantiation: If Tweety is a bird, then Tweety can fly
    if fact_tweety_is_bird:
        result = rule_all_birds_fly("Tweety")
    else:
        result = False

    # Step 4: Modus ponens: Since Tweety is a bird and all birds can fly, conclude Tweety can fly
    if result:
        print("Conclusion: We have proved that Tweety can fly.")
    else:
        print("Conclusion: We could not prove that Tweety can fly.")

# Run the proof
proof_tweety_can_fly()
```

```
Conclusion: We have proved that Tweety can fly.
```

Program 11
Alpha beta Prunning in 8 queens

Algorithm:

Alpha - Beta pruning in 8 - Queens

```
def is_valid(board, row, col):      // checks if it is safe to place
    for r in range(row):              a queen on a particular row
        if board[r] == col or board[r] - r == col - row or
           board[r] + r == col + row:

            return False

    return True

def alpha-beta (board, row, alpha, beta):
    if a queen is placed in all rows we found a solution
    if row == 8:
        return True
    Loop through each column
    for col in range(8):
        if is_valid(board, row, col):
            board[row] = col
    If the current branch cannot result in a better solution
            if alpha >= beta
                break
        Recursively iterate for each row
            if alpha-beta (board, row+1, alpha, beta):
                return True
        Undo current move
        board[row] = -1

    return False
def solve():
    board = [-1] * 8
    alpha = - float('inf')
    beta = float('inf')
    if alpha-beta (board, 0, alpha, beta):
        return board
    else:
        return None
```

O/P:
Solution found
[0, 4, 7, 5, 2, 6, 1, 3]

Code:

```
def is_safe(board, row, col):
    """Check if placing a queen at (row, col) is safe."""
    for i in range(col):
        if board[row][i] == 1:
            return False

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    for i, j in zip(range(row, len(board)), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solve_nqueens_util(board, col):
    """Recursive utility function to solve the 8-Queens Problem."""
    if col >= len(board):  # All queens are placed
        return True

    for i in range(len(board)):
        if is_safe(board, i, col):  # Check if it's safe to place a queen
            board[i][col] = 1

            if solve_nqueens_util(board, col + 1):  # Recur to place the next queen
                return True

            # Backtrack
            board[i][col] = 0

    return False

def solve_nqueens():
    """Solve the 8-Queens Problem and print the solution."""
    n = 8  # Number of queens
    board = [[0] * n for _ in range(n)]

    if not solve_nqueens_util(board, 0):
        print("No solution exists.")
        return

    print("Solution:")
    for row in board:
        print(" ".join("Q" if cell else "." for cell in row))

# Run the solver
solve_nqueens()
```

```
Solution:
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .
```

Program 12

Min Max Algorithm for Tic Tac Toe

Algorithm:

Min Max Algorithm for Tic Tac Toe

def gameover (board):

// check rows, columns, diagonal for a win

for row in range(3):

if board [row][0] == board[row][1] == board [row][2]

and board[row][0] != ' ':

return True

for col in range(3):

if board [0][col] == board[1][col] == board[2][col]

and board[0][col] != ' ':

return True

if board(0)[0] == board[1][1] == board (2)[2] and

board [0][0] != ' ':

check if board is full

for row in range(3):

for col in range(3):

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

return False

return True

Evaluate the board and return the score

def evaluate (board):

if board [row][0] == 'X':

return 10

else:

return -10

return 0

if board [col][0] == 'X':

if board [0][0] == 'X':

if board [0][2] == 'X':

'X': 10

'O': -10

'draw': 0

def minmax (board, depth):
 // Explore all possible future moves evaluate their
Outcomes and choose the best one for the current p

```
 if is_gameOver(board):
     return evaluate (board) //Return score
 Try every possible move along rows
 for row in range(3):
   for col in range(3)
     if board(row)(col) == ' ':
        board(row)(col)  = 'X'

      eval = minimax (board, depth+1, False)
 Try every possible moves along column
else
     if board(row)(col) == ' ':
        board(row)(col) == 'O'

     eval = minimax (board, depth+1, True)
```

Output
current board
X O X
O X O
best move is (2,0)

Received
3/12/24

Code:

```
import math

# Define the board
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

# Check for a winner
def check_winner(board):
    for row in board:
        if row[0] == row[1] == row[2] and row[0] != " ":
            return row[0]

    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != " ":
            return board[0][col]

    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != " ":
        return board[0][0]

    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != " ":
        return board[0][2]

    return None

# Check if there are moves left
def is_moves_left(board):
    for row in board:
        if " " in row:
            return True
    return False

# Minimax algorithm
def minimax(board, depth, is_maximizing):
    winner = check_winner(board)
```

```
    if winner == "X":
        return 10 - depth
    if winner == "O":
        return depth - 10
    if not is_moves_left(board):
        return 0

    if is_maximizing:
        best = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "X"
                    best = max(best, minimax(board, depth + 1, False))
                    board[i][j] = " "
        return best
    else:
        best = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "O"
                    best = min(best, minimax(board, depth + 1, True))
                    board[i][j] = " "
        return best

# Find the best move
def find_best_move(board):
    best_val = -math.inf
    best_move = (-1, -1)

    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = "X"
                move_val = minimax(board, 0, False)
                board[i][j] = " "
                if move_val > best_val:
                    best_val = move_val
```

```
                best_move = (i, j)
    return best_move

# Main game loop
def play_tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    player = "O"  # Player goes first
    print("Tic Tac Toe - You are 'O', Computer is 'X'")
    print_board(board)

    while is_moves_left(board) and check_winner(board) is None:
        if player == "O":
            row, col = map(int, input("Enter your move (row and column): ").split())
            if board[row][col] == " ":
                board[row][col] = "O"
                player = "X"
            else:
                print("Invalid move. Try again.")
        else:
            print("Computer is making a move...")
            row, col = find_best_move(board)
            board[row][col] = "X"
            player = "O"

        print_board(board)

    winner = check_winner(board)
    if winner:
        print(f"The winner is {winner}!")
    else:
        print("It's a draw!")

# Run the game
play_tic_tac_toe()
```

```
Tic Tac Toe - You are 'O', Computer is 'X'
  |   |  
---------
  |   |  
---------
  |   |  
---------
Enter your move (row and column): 0 2
  |   | O
---------
  |   |  
---------
  |   |  
---------
Computer is making a move...
  |   | O
---------
  | X |  
---------
  |   |  
---------
Enter your move (row and column): 1 0
  |   | O
---------
O | X |  
---------
  |   |  
---------
```

```
Computer is making a move...
X |   | O
---------
O | X |
---------
  |   |
---------
Enter your move (row and column): 2 0
X |   | O
---------
O | X |
---------
O |   |
---------
Computer is making a move...
X |   | O
---------
O | X |
---------
O |   | X
---------
The winner is X!
```