VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Samarth Kumar Dubey (1BM22CS235)

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 out by **Samarth Kumar Dubey (1BM22CS235),** 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.

Dr Seema Patil Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Samarth512/AI-lab/tree/main

Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:

Saathi

Lab I

Tic - Tac Toe

Algorithm

1) Initialize the game board:

- Create 3×3 matrix
- Each position in board is initialized to "-"

2) Start the loop
- The game run for a maximum of 9 turns
- The game alternates between 2 players, "X" & "O"

3) Displays the board
- Print current board

4) Player Input
- The current player selects the position
- selected position is checked

5) Update the board
- Place the current player mark in selected position

6) Check for win
- After every move, check if current player has won
- Any row or any column has same mark
  Or either diagonal has same mark

7) Check for draw

Date: 1.10.24

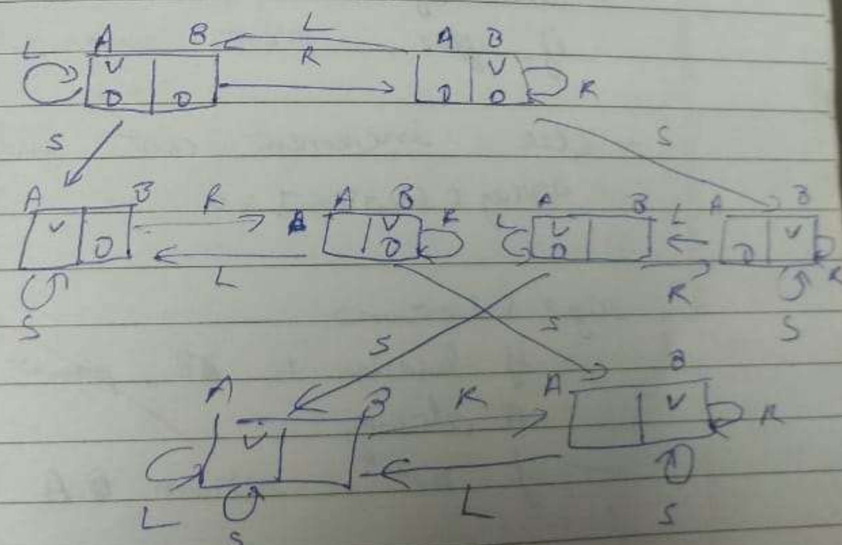
Lab II

Implement vaccume cleaner problem

pseudo code

function : Reflex - Vaccum - Agent ([location, status]) return an action
if status = Dirty then return Suck
else if location = A then return Right
else if location = B then return Left.

The state space diagram



Problem formulation steps

* States
* Initial state
* Actions
* Transition model
* Goal test
* Path cost

Algorithm

1 Initialize variables:
set cost = 0, status array = [1, 1], room identifiers
A = 0, B = 1

2 Functions:
Suck (location)
— check if status for location is clean.
if yes, print room is clean

— else increment cost and and set
array [location] = 0

right (location)
— if location is ~~A~~ B, ~~print~~ cleaner is in
B already
— if not, return ~~B~~ A

left (location)
— if location is A, cleaner is in A
already, return A
— else return A

vaccume cleaner (loc)

3 - check if all rooms are clean, break if true

- check if room is clean, call suck if not

- check location, if A, call right

- check location, call left

1/10/24

Code: 1: Tic - Tac - Toe

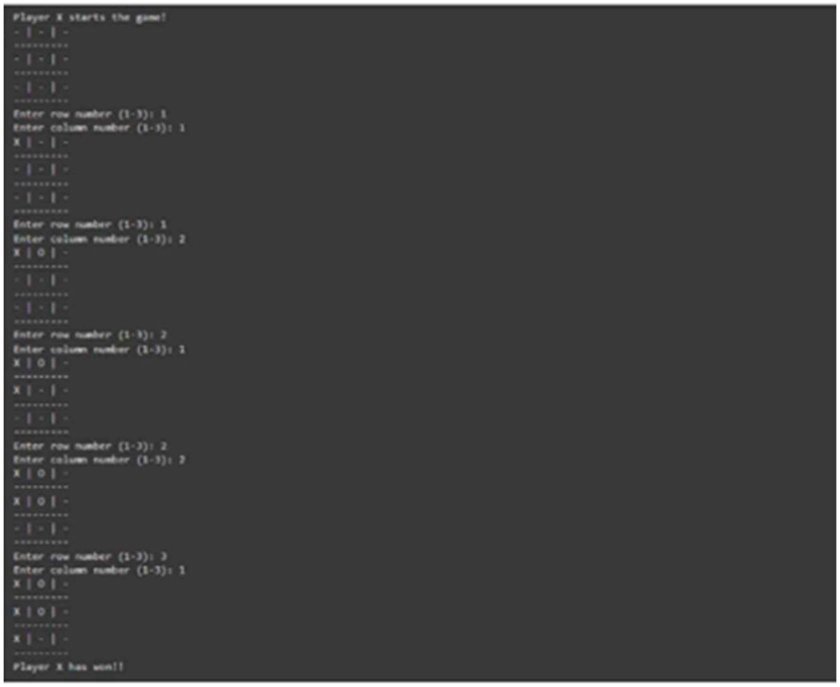
```
import numpy as np

board=np.array([['-','-','-'],['-','-','-'],['-','-','-']])

current_player='X'
flag=0

def check_win():
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != '-':
            return True
    for i in range(3):
        if board[0][i] == board[1][i] == board[2][i] != '-':
            return True
    if board[0][0] == board[1][1] == board[2][2] != '-':
        return True
    if board[0][2] == board[1][1] == board[2][0] != '-':
        return True
    return False

def tic_tac_toe():
    n=0
    print(board)
    while n<9:
        if n%2==0:
            current_player='X'
        else :
            current_player='O'

        row = int(input("Enter row: "))
        col = int(input("Enter column: "))

        if(board[row][col]=='-'):
            board[row][col]=current_player
            print(board)
            flag=check_win();
            if flag==1:
                print(current_player+' wins')
                break
            else:
                n=n+1
        else :
            print("Invalid Position")

    if n==9:
```

    print("Draw")

tic_tac_toe()

Output:

1. WIN



2. DRAW

```
Player X starts the game!
- | - | -
---------
- | - | -
---------
- | - | -
---------
Enter row number (1-3): 1
Enter column number (1-3): 1
X | - | -
---------
- | - | -
---------
- | - | -
---------
Enter row number (1-3): 1
Enter column number (1-3): 3
X | - | O
---------
- | - | -
---------
- | - | -
---------
Enter row number (1-3): 1
Enter column number (1-3): 2
X | X | O
---------
- | - | -
---------
- | - | -
---------
Enter row number (1-3): 2
Enter column number (1-3): 1
X | X | O
---------
O | - | -
---------
- | - | -
---------
Enter row number (1-3): 2
Enter column number (1-3): 3
X | X | O
---------
O | - | X
---------
- | - | -
```

```
Enter row number (1-3): 2
Enter column number (1-3): 3
X | X | O
---------
O | - | X
---------
- | - | -
---------
Enter row number (1-3): 2
Enter column number (1-3): 2
X | X | O
---------
O | O | X
---------
- | - | -
---------
Enter row number (1-3): 3
Enter column number (1-3): 1
X | X | O
---------
O | O | X
---------
X | - | -
---------
Enter row number (1-3): 3
Enter column number (1-3): 2
X | X | O
---------
O | O | X
---------
X | O | -
---------
Enter row number (1-3): 3
Enter column number (1-3): 3
X | X | O
---------
O | O | X
---------
X | O | X
---------
Game is drawn!
```

2. Vacuum Cleaner :

```
cost =0
def vacuum_world(state, location):
  global cost
  if(state['A']==0 and state['B']==0):
    print('All rooms are clean')
    return

  if state[location]==1:
    state[location]=0
    cost+=1
    state[location]=(int(input('Is room '+ str(location) +' still dirty : ')))

    if state[location]==1:
      return vacuum_world(state, location)
    else:
      print('Room ' + str(location) + ' cleaned')

  next_location='B' if location=='A' else 'A'
  if state[next_location]==0:
    state[next_location]=(int(input('Is room '+ str(next_location) +' dirty : ')))
  print('Moving to room '+str(next_location))
  return vacuum_world(state, next_location)

state={}
state['A']=int(input('Enter status of room A : '))
state['B']=int(input('Enter status of room B : '))
location=input('Enter initial location of vacuum (A/B) : ')
vacuum_world(state,location)
print("Status = "+str(state))
print('Total cost: ' + str(cost))
```

Output :

```
Enter status of room A : 1
Enter status of room B : 1
Enter initial location of vacuum (A/B) : A
Is room A still dirty : 0
Room A cleaned
Moving to room B
Is room B still dirty : 0
Room B cleaned
Is room A dirty : 0
Moving to room A
All rooms are clean
Status = {'A': 0, 'B': 0}
Total cost: 2
```

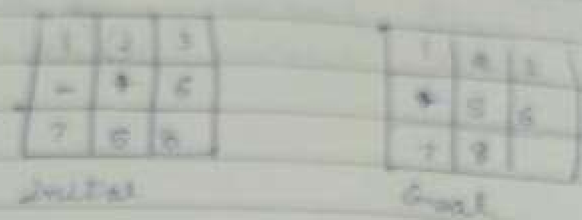
<u>Program 2</u>

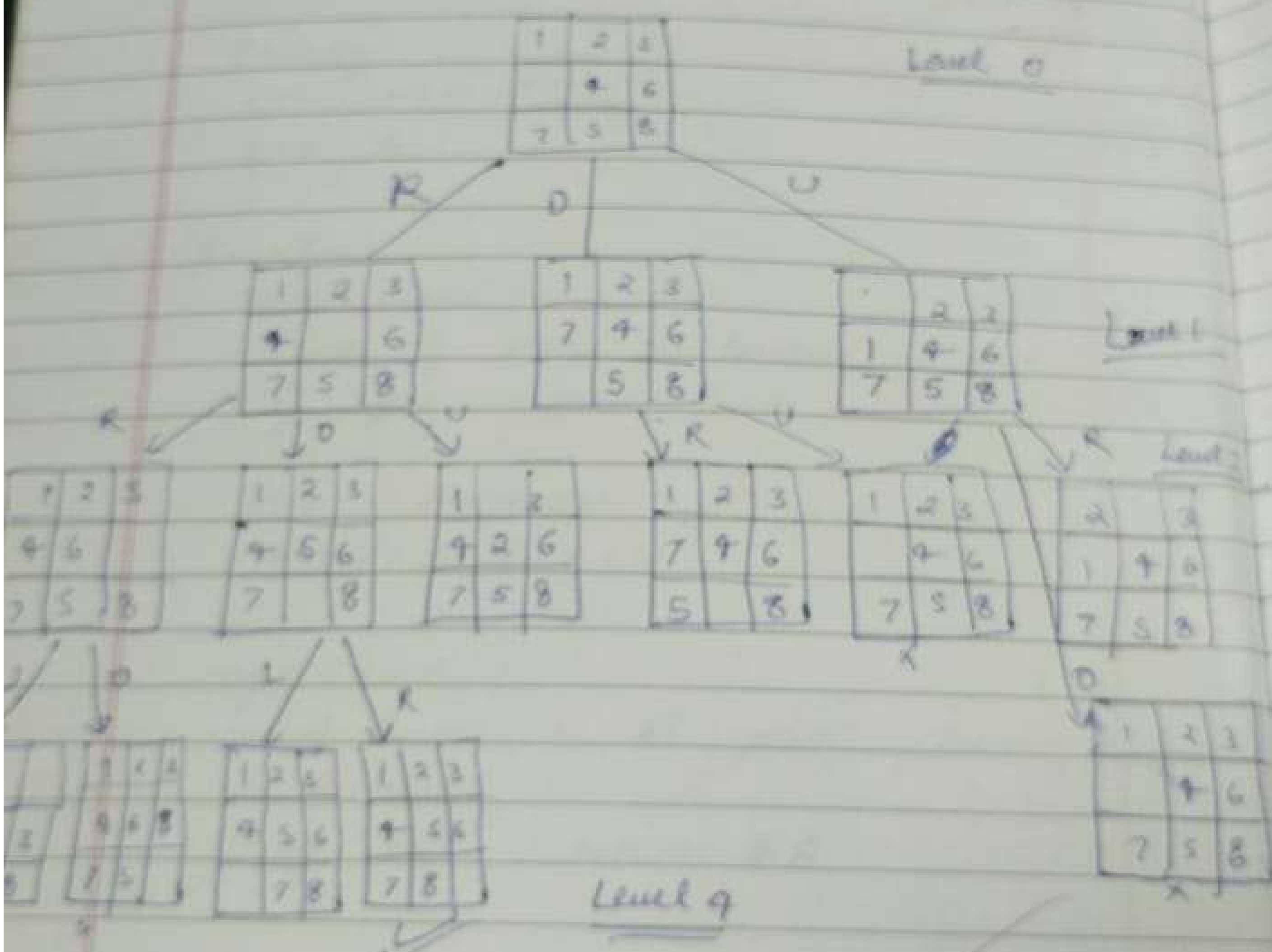
Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

Algorithm:

Lab III

Implement algorithm to solve 8 puzzle problem using BFS & DFS





Date

Algorithm (BFS)

1 Define puzzle class with a 2D array for board, empty tile-position to track empty tile's position and moves a list to keep track of actions taken

2 is_goal(): to check if current state matches goal
move(): to check possible moves
to_string(): to make a state into a string to make sure it is visited

3 BFS ~~loop~~ initialization

Create a function BFS to accept initial state, add the state to queue.

4 While queue is not empty.
i) dequeue from ~~~~ front
ii) Check current states is goal state using is_goal(). if yes, break
iii) Convert current state to string and store in visited

5 After the loop, print total unique states.

Lab IV

1. Iterative deepening of search algorithm using 8 puzzle

a) For each child of current node
b) If it is target node, return
c) If max depth is reached, return
d) Set current node and go back to 1
e) After having gone through all children, go to the next child of the parent (next sibling)
f) After having gone through all children, go to next child & increase max depth and go to a
g) If we have reached all leaf nodes, the goal node doesn't exist.

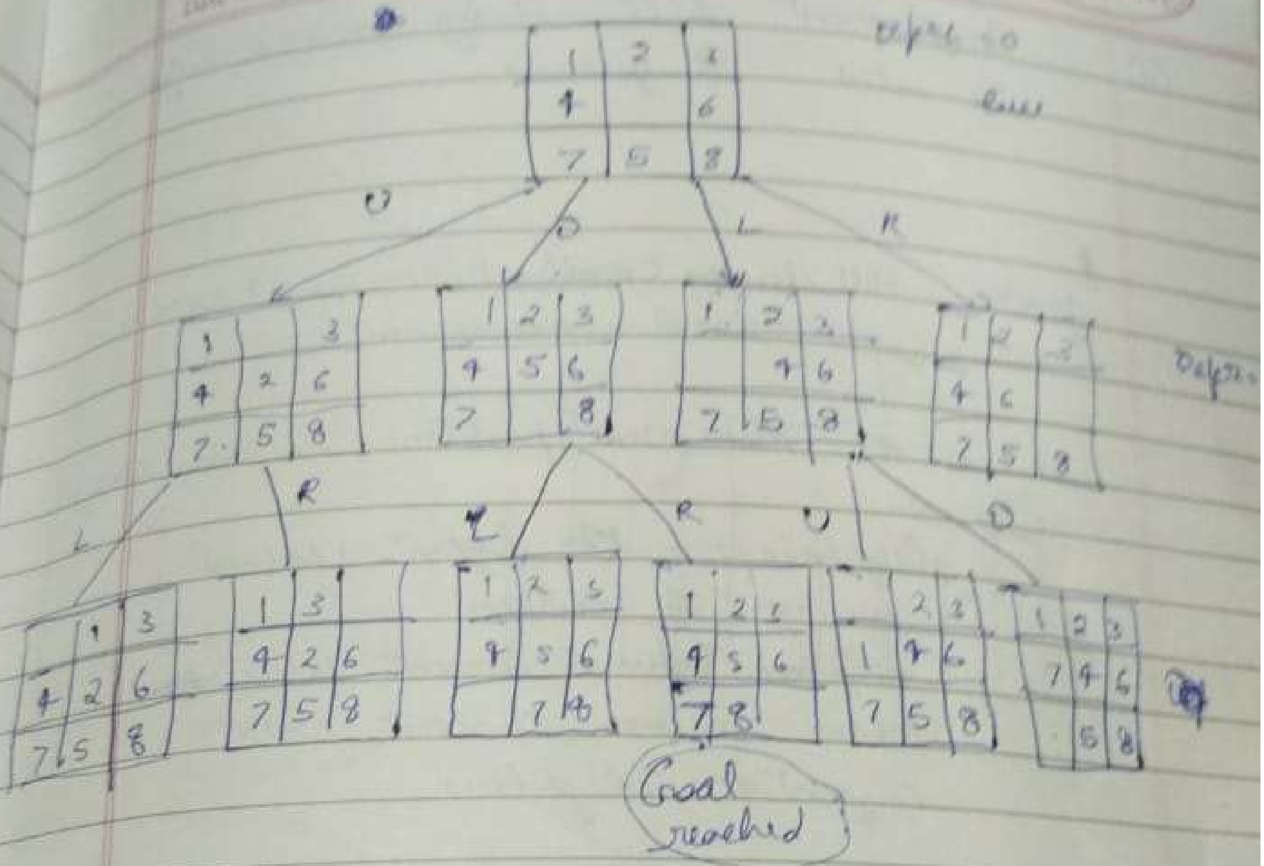
State space diagram

1	2	3
4		6
7	5	8

Initial

1	2	3
4	5	6
7	8	

Goal State



Depth = 2

5-12

Code:
1: DFS

```
cnt = 0;
def print_state(in_array):
 global cnt
 cnt += 1
 for row in in_array:
  print(' '.join(str(num) for num in row))
 print()

def helper(goal, in_array, row, col, vis):

 vis[row][col] = 1
 drow = [-1, 0, 1, 0]
 dcol = [0, 1, 0, -1]
 dchange = ['U', 'R', 'D', 'L']

 print("Current state:")
 print_state(in_array)

 if in_array == goal:
  print_state(in_array)
  print(f"Number of states : {cnt}")
  return True


 for i in range(4):
  nrow = row + drow[i]
  ncol = col + dcol[i]

  if 0 <= nrow < len(in_array) and 0 <= ncol < len(in_array[0]) and not vis[nrow][ncol]:

    print(f"Took a {dchange[i]} move")
    in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol], in_array[row][col]

    if helper(goal, in_array, nrow, ncol, vis):
      return True

    in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol], in_array[row][col]


 vis[row][col] = 0
 return False


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

```
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
visited = [[0] * 3 for _ in range(3)]
empty_row, empty_col = 1, 0
found_soluΘon = helper(goal_state, iniΘal_state, empty_row, empty_col, visited)
print("SoluΘon found:", found_soluΘon)
```

Output :

```
Current state:
1 2 3
0 4 6
7 5 8

Took a U move
Current state:
0 2 3
1 4 6
7 5 8

Took a R move
Current state:
2 0 3
1 4 6
7 5 8

Took a R move
Current state:
2 3 0
1 4 6
7 5 8

Took a D move
Current state:
2 3 6
1 4 0
7 5 8

Took a D move
Current state:
2 3 6
1 4 8
7 5 0

Took a L move
Current state:
2 3 6
1 4 8
7 0 5

Took a U move
Current state:
2 3 6
1 0 8
7 4 5

Took a L move
Current state:
2 3 6
1 4 8
0 7 5

Took a L move
Current state:
2 3 6
1 0 4
7 5 8

Took a D move
Current state:
2 3 6
1 5 4
7 0 8

Took a R move
Current state:
2 3 6
1 5 4
7 8 0

Took a L move
Current state:
2 3 6
1 5 4
0 7 8

Took a D move
Current state:
2 4 3
1 0 6
7 5 8

Took a R move
Current state:
2 4 3
1 6 0
7 5 8

Took a U move
Current state:
2 4 0
1 6 3
7 5 8

Took a L move
Current state:
1 0 2
4 6 3
7 5 8

Took a L move
Current state:
0 1 2
4 6 3
7 5 8

Took a D move
Current state:
1 2 3
4 6 8
7 5 0

Took a L move
Current state:
1 2 3
4 6 8
7 0 5

Took a L move
Current state:
1 2 3
4 6 8
0 7 5

Took a D move
Current state:
1 2 3
4 5 6
7 0 8

Took a R move
Current state:
1 2 3
4 5 6
7 8 0

1 2 3
4 5 6
7 8 0

Number of states : 42
Soluθon found: True
```

2 : Iterative deepening search

```
class PuzzleState:
    def __init__(self, board, empty_tile_pos, depth=0, path=[]):
        self.board = board
        self.empty_tile_pos = empty_tile_pos  # (row, col)
        self.depth = depth
        self.path = path  # Keep track of the path taken to reach this state

    def is_goal(self, goal):
        return self.board == goal

    def generate_moves(self):
        row, col = self.empty_tile_pos
        moves = []
        directions = [(-1, 0, 'Up'), (1, 0, 'Down'), (0, -1, 'Left'), (0, 1, 'Right')]  # up, down, left, right
        for dr, dc, move_name in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] = new_board[new_row *
3 + new_col], new_board[row * 3 + col]
                new_path = self.path + [move_name]  # Update the path with the new move
                moves.append(PuzzleState(new_board, (new_row, new_col), self.depth + 1, new_path))
        return moves

    def display(self):
        # Display the board in a matrix form
        for i in range(0, 9, 3):
            print(self.board[i:i + 3])
        print(f"Moves: {self.path}")  # Display the moves taken to reach this state
        print()  # Newline for better readability

def iddfs(initial_state, goal, max_depth):
    for depth in range(max_depth + 1):
        print(f"Searching at depth: {depth}")
        found = dls(initial_state, goal, depth)
        if found:
            print(f"Goal found at depth: {found.depth}")
            found.display()
            return found
    print("Goal not found within max depth.")
    return None

def dls(state, goal, depth):
    if state.is_goal(goal):
        return state
```

```
    if depth <= 0:
        return None

    for move in state.generate_moves():
        print("Current state:")
        move.display()  # Display the current state
        result = dls(move, goal, depth - 1)
        if result is not None:
            return result
    return None

def main():
    # User input for initial state, goal state, and maximum depth
    initial_state_input = input("Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8
0'): ")
    goal_state_input = input("Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8
0'): ")
    max_depth = int(input("Enter maximum depth: "))

    initial_board = list(map(int, initial_state_input.split()))
    goal_board = list(map(int, goal_state_input.split()))
    empty_tile_pos = initial_board.index(0) // 3, initial_board.index(0) % 3  # Calculate the position of
the empty tile

    initial_state = PuzzleState(initial_board, empty_tile_pos)

    solution = iddfs(initial_state, goal_board, max_depth)

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

Output :

```
Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1 2 3 0 4 6 7 5 8
Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1 2 3 4 5 6 7 8 0
Enter maximum depth: 2
Searching at depth: 0
Searching at depth: 1

Current state:
[0, 2, 3]
[1, 4, 6]
[7, 5, 8]
Moves: ['Up']

Current state:
[1, 2, 3]
[7, 4, 6]
[0, 5, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
Moves: ['Right']

Searching at depth: 2
Current state:
[0, 2, 3]
[1, 4, 6]
[7, 5, 8]
Moves: ['Up']

Current state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Moves: ['Up', 'Down']

Current state:
[2, 0, 3]
[1, 4, 6]
[7, 5, 8]
Moves: ['Up', 'Right']

Current state:
[1, 2, 3]
[7, 4, 6]
[0, 5, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[7, 4, 6]
[0, 5, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Moves: ['Down', 'Up']

Current state:
[1, 2, 3]
[7, 4, 6]
[5, 0, 8]
Moves: ['Down', 'Right']

Current state:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
Moves: ['Right']

Current state:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]
Moves: ['Right', 'Up']

Current state:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Moves: ['Right', 'Down']

Current state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Moves: ['Right', 'Left']

Current state:
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]
Moves: ['Right', 'Right']

Goal not found within max depth.
```

Program 3

Implement A* search algorithm

Algorithm:

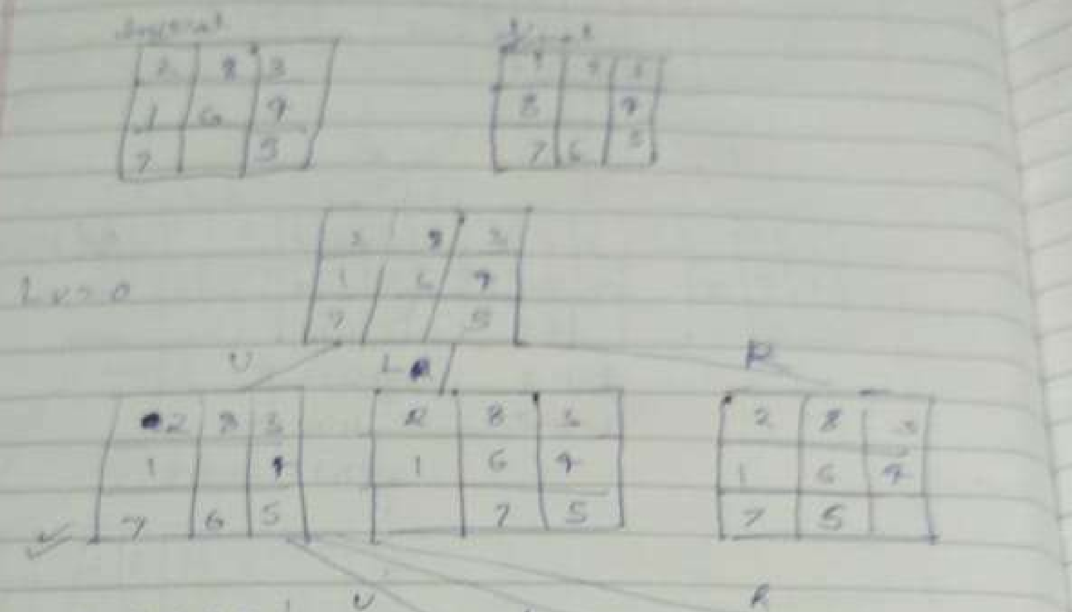
Algorithm : Misplaced tiles

1 Record initial and ~~final~~ goal states as required.
2 Push starting state in open list, then check possible moves.
3 For each move, choose the move which will produce the state with least mismatched tiles by comparing with goal.
4 For each move, increment the level counter.
5 Go to step 3 until current state matches goal state.
6 Once goal state is achieved, return cost as (levels + mismatched tiles).

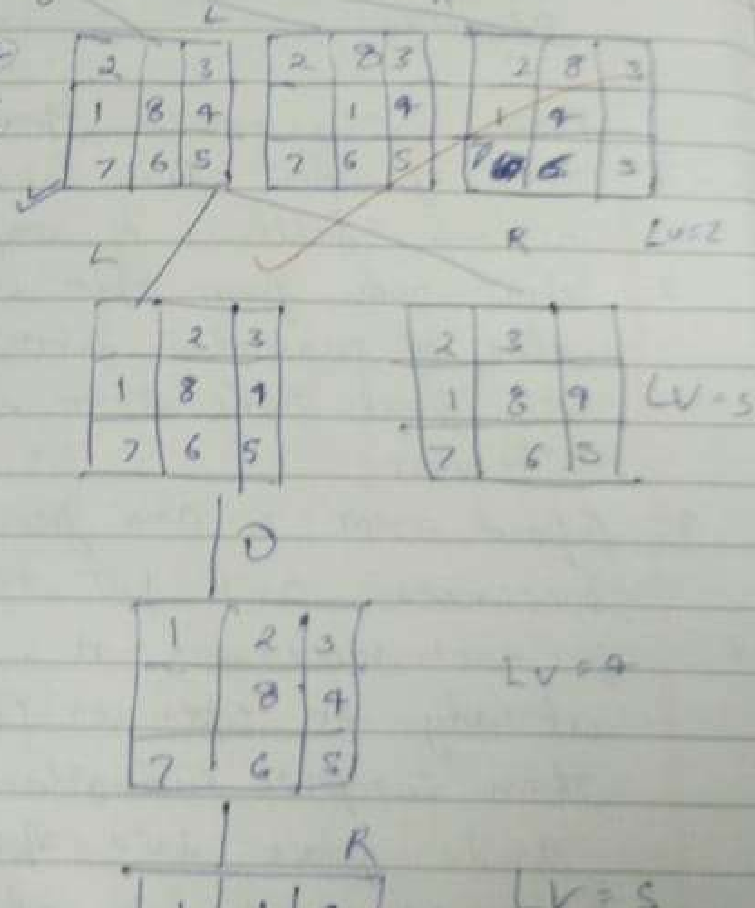
Algorithm : Manhattan distance

1 Record initial and goal state as required
2 Push starting state in open list, then check possible moves
3 For each possible move, check the manhattan distance
4 calculate manhattan distance by calculating minimum number of moves to reach goal state, and add them.
5 Choose the ~~minimum~~ minimum distance state, push in open while ~~removing~~ popping previous state.

b) Using manhattan distance



M.D	1	2	3	4	5	6	7	8	
U	1	1	0	0	0	0	0	2	4
R	1	1	0	0	1	1	0	2	6
L	1	1	0	0	0	1	1	2	6



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

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

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

Code:

Misplaced Tiles :

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

    def heuristic(self):
        goal_state = [[1,2,3], [8,0,4], [7,6,5]]
        count = 0
        for i in range(len(self.state)):
            for j in range(len(self.state[i])):
                if self.state[i][j] != 0 and self.state[i][j] != goal_state[i][j]:
                    count += 1
        return count

def get_blank_position(state):
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                return i, j

def get_possible_moves(position):
    x, y = position
    moves = []
    if x > 0: moves.append((x - 1, y, 'Down'))
    if x < 2: moves.append((x + 1, y, 'Up'))
    if y > 0: moves.append((x, y - 1, 'Right'))
    if y < 2: moves.append((x, y + 1, 'Left'))
    return moves

def generate_new_state(state, blank_pos, new_blank_pos):
    new_state = [row[:] for row in state]
    new_state[blank_pos[0]][blank_pos[1]], new_state[new_blank_pos[0]][new_blank_pos[1]] = \
        new_state[new_blank_pos[0]][new_blank_pos[1]], new_state[blank_pos[0]][blank_pos[1]]
    return new_state

def a_star_search(initial_state):
    open_list = []
    closed_list = set()

    initial_node = Node(state=initial_state, cost=0)
```

```
    open_list.append(initial_node)

    while open_list:

        open_list.sort(key=lambda node: node.cost + node.heuristic())
        current_node = open_list.pop(0)


        move_description = current_node.move if current_node.move else "Start"
        print("Current state:")
        for row in current_node.state:
            print(row)
        print(f"Move: {move_description}")
        print(f"Heuristic value (misplaced tiles): {current_node.heuristic()}")
        print(f"Cost to reach this node: {current_node.cost}\n")

        if current_node.heuristic() == 0:

            path = []
            while current_node:
                path.append(current_node)
                current_node = current_node.parent
            return path[::-1]
        closed_list.add(tuple(map(tuple, current_node.state)))

        blank_pos = get_blank_position(current_node.state)
        for new_blank_pos in get_possible_moves(blank_pos):
            new_state = generate_new_state(current_node.state, blank_pos, (new_blank_pos[0],
new_blank_pos[1]))

            if tuple(map(tuple, new_state)) in closed_list:
                continue

            cost = current_node.cost + 1
            move_direction = new_blank_pos[2]
            new_node = Node(state=new_state, parent=current_node, move=move_direction, cost=cost)

            if new_node not in open_list:
                open_list.append(new_node)

    return None

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
solution_path = a_star_search(initial_state)

if solution_path:
    print("Solution path:")
```

```
    for step in solution_path:
        for row in step.state:
            print(row)
        print()
else:
    print("No solution found.")
```

\

Output :

```
Current state:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Move: Start
Heuristic value (misplaced tiles): 4
Cost to reach this node: 0

Current state:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Move: Down
Heuristic value (misplaced tiles): 3
Cost to reach this node: 1

Current state:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Move: Down
Heuristic value (misplaced tiles): 3
Cost to reach this node: 2

Current state:
[2, 8, 3]
[0, 1, 4]
[7, 6, 5]
Move: Right
Heuristic value (misplaced tiles): 3
Cost to reach this node: 2

Current state:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
Move: Right
Heuristic value (misplaced tiles): 2
Cost to reach this node: 3

Current state:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
Move: Up
Heuristic value (misplaced tiles): 1
Cost to reach this node: 4

Current state:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
Move: Left
Heuristic value (misplaced tiles): 0
Cost to reach this node: 5

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

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

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

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

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

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

Code :

Manhattan distance approach

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

    def heuristic(self):
        goal_positions = {
            1: (0, 0), 2: (0, 1), 3: (0, 2),
            8: (1, 0), 0: (1, 1), 4: (1, 2),
            7: (2, 0), 6: (2, 1), 5: (2, 2)
        }
        manhattan_distance = 0
        for i in range(len(self.state)):
            for j in range(len(self.state[i])):
                value = self.state[i][j]
                if value != 0:
                    goal_i, goal_j = goal_positions[value]
                    manhattan_distance += abs(i - goal_i) + abs(j - goal_j)
        return manhattan_distance

def get_blank_position(state):
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                return i, j

def get_possible_moves(position):
    x, y = position
    moves = []
    if x > 0: moves.append((x - 1, y, 'Down'))
    if x < 2: moves.append((x + 1, y, 'Up'))
    if y > 0: moves.append((x, y - 1, 'Right'))
    if y < 2: moves.append((x, y + 1, 'Left'))
    return moves

def generate_new_state(state, blank_pos, new_blank_pos):
    new_state = [row[:] for row in state]
    new_state[blank_pos[0]][blank_pos[1]], new_state[new_blank_pos[0]][new_blank_pos[1]] = \
        new_state[new_blank_pos[0]][new_blank_pos[1]], new_state[blank_pos[0]][blank_pos[1]]
    return new_state

def a_star_search(initial_state):
    open_list = []
```

```
    closed_list = set()

    initial_node = Node(state=initial_state, cost=0)
    open_list.append(initial_node)

    while open_list:

        open_list.sort(key=lambda node: node.cost + node.heuristic())
        current_node = open_list.pop(0)


        move_description = current_node.move if current_node.move else "Start"
        print("Current state:")
        for row in current_node.state:
            print(row)
        print(f"Move: {move_description}")
        print(f"Heuristic value (Manhattan distance): {current_node.heuristic()}")
        print(f"Cost to reach this node: {current_node.cost}\n")

        if current_node.heuristic() == 0:

            path = []
            while current_node:
                path.append(current_node)
                current_node = current_node.parent
            return path[::-1]
        closed_list.add(tuple(map(tuple, current_node.state)))

        blank_pos = get_blank_position(current_node.state)
        for new_blank_pos in get_possible_moves(blank_pos):
            new_state = generate_new_state(current_node.state, blank_pos, (new_blank_pos[0],
new_blank_pos[1]))

            if tuple(map(tuple, new_state)) in closed_list:
                continue

            cost = current_node.cost + 1
            move_direction = new_blank_pos[2]
            new_node = Node(state=new_state, parent=current_node, move=move_direction, cost=cost)

            if new_node not in open_list:
                open_list.append(new_node)

    return None

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
solution_path = a_star_search(initial_state)
```

```
if solution_path:
    print("Solution path:")
    for step in solution_path:
        for row in step.state:
            print(row)
        print()
else:
    print("No solution found.")
```

Output :

```
Current state:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Move: Start
Heuristic value (Manhattan distance): 5
Cost to reach this node: 0

Current state:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Move: Down
Heuristic value (Manhattan distance): 4
Cost to reach this node: 1

Current state:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Move: Down
Heuristic value (Manhattan distance): 3
Cost to reach this node: 2

Current state:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
Move: Right
Heuristic value (Manhattan distance): 2
Cost to reach this node: 3
```

```
Current state:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
Move: Left
Heuristic value (Manhattan distance): 0
Cost to reach this node: 5

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

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

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

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

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

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

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

Date ______

② Implement hill climbing search algo. to solve 8 queens.

function Hill climbing (problem) returns a state that is local maximum

current ← MakeNode(problem. Initial-state)

loop do

neighbour ← the highest valued successor of current

if neighbour. Value ≤ Current. Value ~~stem~~ then return current. state

current ← neighbour

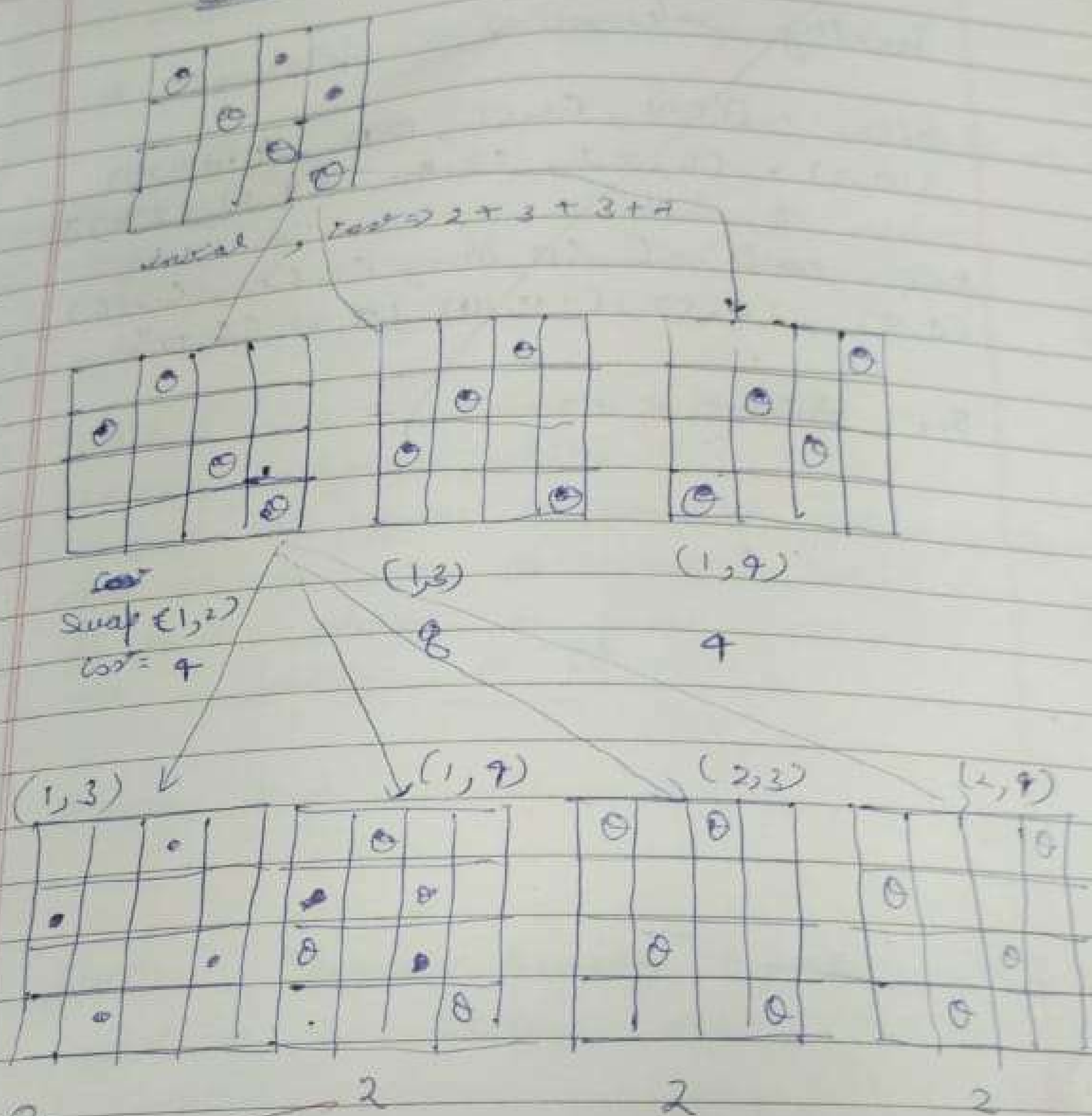
State: 4 queens on the board. One queen per column

- variables - x_0, x_1, x_2, x_3 where x_i is row position of queen in column i
- Domain for each $x_i \in \{0, 1, 2, 3\}$. $\forall i$

Initial state: a random state

Goal state: 4 queens on the board,

State space



Goal reached

Code:

```
import random
def calculate_cost(board):

    n = len(board)

    attacks = 0

    for i in range(n):

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

            if board[i] == board[j]:  # Same column

                attacks += 1

            if abs(board[i] - board[j]) == abs(i - j):  # Same diagonal

                attacks += 1

    return attacks


def get_neighbors(board):

    neighbors = []

    n = len(board)

    for col in range(n):

        for row in range(n):

            if row != board[col]:  # Only change the row of the queen

                new_board = board[:]

                new_board[col] = row

                neighbors.append(new_board)

    return neighbors
```

```
def hill_climb(board, max_restarts=100):
    current_cost = calculate_cost(board)
    print("Initial board configuration:")
    print_board(board, current_cost)
    iteration = 0
    restarts = 0

    while restarts < max_restarts:  # Add limit to the number of restarts
        while current_cost != 0:  # Continue until cost is zero
            neighbors = get_neighbors(board)
            best_neighbor = None
            best_cost = current_cost

            for neighbor in neighbors:
                cost = calculate_cost(neighbor)
                if cost < best_cost:  # Looking for a lower cost
                    best_cost = cost
                    best_neighbor = neighbor

            if best_neighbor is None:  # No better neighbor found
                break  # Break the loop if we are stuck at a local minimum

            board = best_neighbor
```

```
            current_cost = best_cost
            iteration += 1
            print(f"Iteration {iteration}:")
            print_board(board, current_cost)

        if current_cost == 0:
            break  # We found the solution, no need for further restarts
        else:
            # Restart with a new random configuration
            board = [random.randint(0, len(board)-1) for _ in range(len(board))]
            current_cost = calculate_cost(board)
            restarts += 1
            print(f"Restart {restarts}:")
            print_board(board, current_cost)

    return board, current_cost


def print_board(board, cost):
    n = len(board)
    display_board = [['.'] * n for _ in range(n)]  # Create an empty board
    for col in range(n):
        display_board[board[col]][col] = 'Q'  # Place queens on the board
    for row in range(n):
        print(' '.join(display_board[row]))  # Print the board
```

```python
    print(f"Cost: {cost}\n")


if __name__ == "__main__":
    n = int(input("Enter the number of queens (N): "))  # User input for N
    initial_state = list(map(int, input(f"Enter the initial state (row numbers for each column, space-separated): ").split()))
    if len(initial_state) != n or any(r < 0 or r >= n for r in initial_state):
        print("Invalid initial state. Please ensure it has N elements with values from 0 to N-1.")
    else:
        solution, cost = hill_climb(initial_state)
        if cost == 0:
            print(f"Solution found with no conflicts:")
        else:
            print(f"No solution found within the restart limit:")
        print_board(solution, cost)
```

Output :

```
Enter the number of queens (N): 4
Enter the initial state (row numbers for each column, space-separated): 0 1 2 3
Initial board configuration:
```

```
Q . . .
. Q . .
. . Q .
. . . Q
Cost: 6

Iteration 1:
. . . .
Q Q . .
. . Q .
. . . Q
Cost: 4

Iteration 2:
. Q . .
Q . . .
. . Q .
. . . Q
Cost: 2

Restart 1:
. Q Q Q
. . . .
. . . .
Q . . .
Cost: 4

Iteration 3:
. Q . Q
. . . .
. . Q .
Q . . .
Cost: 2

Iteration 4:
. Q . .
. . . Q
. . Q .
Q . . .
Cost: 1

Restart 2:
. . . Q
. Q . .
. . . .
Q . Q .
Cost: 2

Iteration 6:
. . . .
. Q . .
. . Q Q
Q . . .
Cost: 2

Iteration 7:
. . Q .
. Q . .
. . . Q
Q . . .
Cost: 1

Restart 4:
Q . . .
. Q . Q
. . Q .
. . . .
Cost: 5

Iteration 8:
Q . . .
. Q . Q
. . . .
. . Q .
Cost: 2

Iteration 9:
Q Q . .
. . . Q
. . . .
. . Q .
Cost: 1

Iteration 10:
. Q . .
. . . Q
Q . . .
. . Q .
Cost: 0

Solution found with no conflicts:
. Q . .
. . . Q
Q . . .
. . Q .
Cost: 0
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

Lab 5

Simulated annealing algorithm

1) Initialize some solution & temperature T.
2) Define a cooling schedule to reduce temperature over time and a stopping criterion

3) while Temp > minimum Temperature
a) generate neighbour solution
b) Compute cost difference compared to initial solution and choose better as current solution
c) If new solution is worse, accept it with probability $e^{-\frac{\Delta E}{KT}}$

4) Return final solution

function Simulated Annealing (problem Schedule) returns a solution state

Input => problem

~~current ←~~ Make-Node
S ← initial_solution
T ← initial_temperature
while T > min_temperature
S_new = generate_neighbour(S)
[illegible] = cost(S_new) - cost(S)

T = cooling_schedule (T)

return s

Example
Travelling salesman

cities ⇒ [(0,0), (1,5), (5,1), (10,10),
(10,5), (6,7), (3,8), (8,2), (2,6)]

Best route = [(0,0), (1,5), (2,6), (3,8),
(6,5), (10,10), (8,2), (5,1)]

Best distance = 33

3-12-2

Code:

```
#!pip install mlrose-hiive joblib
#!pip install --upgrade joblib
#!pip install joblib==1.1.0
import mlrose_hiive as mlrose
import numpy as np

def queens_max(position):
    no_attack_on_j = 0
    queen_not_attacking = 0
    for i in range(len(position) - 1):
        no_attack_on_j = 0
        for j in range(i + 1, len(position)):
            if (position[j] != position[i]) and (position[j] != position[i] + (j - i)) and (position[j] !=
            position[i] - (j - i)):
                no_attack_on_j += 1
        if (no_attack_on_j == len(position) - 1 - i):
            queen_not_attacking += 1
    if (queen_not_attacking == 7):
        queen_not_attacking += 1
    return queen_not_attacking

objective = mlrose.CustomFitness(queens_max)

problem = mlrose.DiscreteOpt(length=8, fitness_fn=objective, maximize=True, max_val=8)
T = mlrose.ExpDecay()

initial_position = np.array([4, 6, 1, 5, 2, 0, 3, 7])

#The simulated_annealing function returns 3 values, we need to capture all 3
best_position, best_objective, fitness_curve = mlrose.simulated_annealing(problem=problem,
schedule=T, max_attempts=500,
                                                init_state=initial_position)

print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
```

Output :

```
The best position found is: [4 0 7 5 2 6 1 3]
The number of queens that are not attacking each other is: 8.0
```

Program 6

Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm:

Date

Lab 6

Truth table enumeration algorithm for deciding propositional entailment

A truth table enumeration algorithm for deciding propositional entailment. PL-True? returns true if a sentence holds within a model. The variable model represents a partial model - an assignment to some of the symbols. The keyword "and" is used here as a logical operation on its true arguments, returning true / false.

function TT-Entails (KB, α) returns true or false
inputs: KB, the knowledge base, a sentence
α ⇒ query, a sentence in propositional logic

symbols ← list of proposition symbols in KB and α.
return TT-check-All (KB, α, symbols, { })

function TT-check-All (KB, α, symbols, model) return true or false.
if Empty ! (symbols) then
if PL-True (KB, model) then

$p \leftarrow$ First (Symbols)
rest $\leftarrow$ Rest (Symbols)
return (TT-check-All (KB, α,
rest, model $\cup$ {p = True})
and
TT-checkAll (KB, α, rest,
model $\cup$ {p = False})

Example Case :-

$KB \models \alpha$

$\alpha \Rightarrow A \vee B$, $KB \Rightarrow (A \vee C) \wedge (B \vee \neg C)$

A	B	C	$A \vee C$	$B \vee \neg C$	KB	α
false	false	false	false	True	false	false
false	false	True	True	false	false	false
false	True	false	false	True	false	True
false	True	True	True	~~false~~ True	True	True
True	false	false	True	True	True	True
True	false	True	True	false	false	True
True	True	false	True	True	True	True
True	True	True	True	True	True	True

O/P seen

12/11/24

Code:

```
import pandas as pd

# Define the truth table for all combinations of A, B, C
truth_values = [(False, False, False),
                (False, False, True),
                (False, True, False),
                (False, True, True),
                (True, False, False),
                (True, False, True),
                (True, True, False),
                (True, True, True)]

# Columns: A, B, C
table = pd.DataFrame(truth_values, columns=["A", "B", "C"])

# Calculate intermediate columns
table["A or C"] = table["A"] | table["C"]        # A ∨ C
table["B or not C"] = table["B"] | ~table["C"]   # B ∨ ¬C

# Knowledge Base (KB): (A ∨ C) ∧ (B ∨ ¬C)
table["KB"] = table["A or C"] & table["B or not C"]

# Alpha (α): A ∨ B
table["Alpha (α)"] = table["A"] | table["B"]

# Define a highlighting function
def highlight_rows(row):
    if row["KB"] and row["Alpha (α)"]:
        return ['font-weight: bold; color: black'] * len(row)
    else:
        return [''] * len(row)

# Apply the highlighting function
styled_table = table.style.apply(highlight_rows, axis=1)

# Display the styled table
styled_table
```

Output :

	A	B	C	A or C	B or not C	KB	Alpha (α)
0	False	False	False	False	True	False	False
1	False	False	True	True	False	False	False
2	False	True	False	False	True	False	True
3	**False**	**True**	**True**	**True**	**True**	**True**	**True**
4	**True**	**False**	**False**	**True**	**True**	**True**	**True**
5	True	False	True	True	False	False	True
6	**True**	**True**	**False**	**True**	**True**	**True**	**True**
7	**True**	**True**	**True**	**True**	**True**	**True**	**True**

Program 7

Implement unification in first order logic

Algorithm:

Lab 7

Implement unification in FOL

Algorithm
Unify (Ψ_1 , Ψ_2)
Step 1 : If Ψ_1 or Ψ_2 is a variable or constant then :
a) If Ψ_1 or Ψ_2 are identical, then return NIL
b) If Ψ_1 is a variable
a) then if Ψ_1 occurs in Ψ_2 , then return failure
b) Else return { (Ψ_2 / Ψ_1) }

$\Psi_1 \neq$ ... $\Psi_2 \neq$ ...
c) Else if Ψ_2 is a variable
a) If Ψ_2 occurs in Ψ_1 then return ~~failure~~ FAILURE
b) Else return{ (Ψ_1 / Ψ_2) }

Step 2 : If the initial predicate symbol in Ψ_1 & Ψ_2 are not same , then return FAILURE

Step 3 : If Ψ_1 & Ψ_2 have different number of arguments , then return FAILURE

Step 4 : Set substitution set (SUBST) to NIL

Step 5 : For i=1 to the number of elements and put the result into S.

b) If S = failure then return FAILURE

c) If S ≠ NIL then do
a) Apply S to the remainder to both L1 & L2
b) SUBST = APPEND(S, SUBST)

step 6 : Return SUBST

6/9/21

Generalized Modus Ponens

Forward chaining

Ex => $p_1' = $ Faster (Bob, Pat)
$p_2' = $ Faster (Pat, Steve)

$p_1 \wedge p_2 \Rightarrow q$ = Faster (x, y) ∧ Faster (y, z)
⇒ Faster (x, z)

$\sigma = \{x/Bob, y/Pat, z/Steve\}$
$q\sigma$ = Faster (Bob, Steve)

Example : $\varphi_1 = f(X, Y)$, $\varphi_2 = f(a, b)$

Saathi

5) Start a loop and call i-th element of both ~~a~~ expressions
E.g. unify (x, a) ⇒ ~~a~~

6) ~~Add~~ ~~one~~ After each return from ~~parameter~~ function add to set
Subst = $\{x : a,\ y : b\}$

7) Return subst

Problems

Q. $\psi_1 = P(f(a),\ g(y))$
$\psi_2 = P(x,\ x)$

Q. $\psi_1 = P(b,\ x,\ f(g(z)))$
$\psi_2 = P(z,\ f(y),\ f(y))$

A. failure as x & $\underline{f(a)}$ don't have same predicate symbol
~~initial~~

A. for $i = 0$
$x : f(a)$
for $i = 1$
$\underline{f}(a) : \underline{g}(y)$

A. for ~~i~~ P is same, 3 argument

Code:
```
import re

def occurs_check(var, x):
    """Checks if var occurs in x (to prevent circular substitutions)."""
    if var == x:
        return True
    elif isinstance(x, list):  # If x is a compound expression (like a function or predicate)
        return any(occurs_check(var, xi) for xi in x)
    return False


def unify_var(var, x, subst):
    """Handles unification of a variable with another term."""
    if var in subst:  # If var is already substituted
        return unify(subst[var], x, subst)
    elif isinstance(x, (list, tuple)) and tuple(x) in subst:  # Handle compound expressions
        return unify(var, subst[tuple(x)], subst)
    elif occurs_check(var, x):  # Check for circular references
        return "FAILURE"
    else:
        # Add the substitution to the set (convert list to tuple for hashability)
        subst[var] = tuple(x) if isinstance(x, list) else x
        return subst


def unify(x, y, subst=None):
    """
    Unifies two expressions x and y and returns the substitution set if they can be unified.
    Returns 'FAILURE' if unification is not possible.
    """
    if subst is None:
        subst = {}  # Initialize an empty substitution set

    # Step 1: Handle cases where x or y is a variable or constant
    if x == y:  # If x and y are identical
        return subst
    elif isinstance(x, str) and x.islower():  # If x is a variable
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower():  # If y is a variable
        return unify_var(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list):  # If x and y are compound expressions (lists)
        if len(x) != len(y):  # Step 3: Different number of arguments
            return "FAILURE"

        # Step 2: Check if the predicate symbols (the first element) match
        if x[0] != y[0]:  # If the predicates/functions are different
```

```
            return "FAILURE"

        # Step 5: Recursively unify each argument
        for xi, yi in zip(x[1:], y[1:]):  # Skip the predicate (first element)
            subst = unify(xi, yi, subst)
            if subst == "FAILURE":
                return "FAILURE"
        return subst
    else:  # If x and y are different constants or non-unifiable structures
        return "FAILURE"


def unify_and_check(expr1, expr2):
    """
    Attempts to unify two expressions and returns a tuple:
    (is_unified: bool, substitutions: dict or None)
    """
    result = unify(expr1, expr2)
    if result == "FAILURE":
        return False, None
    return True, result


def display_result(expr1, expr2, is_unified, subst):
    print("Expression 1:", expr1)
    print("Expression 2:", expr2)
    if not is_unified:
        print("Result: Unification Failed")
    else:
        print("Result: Unification Successful")
        print("Substitutions:", {k: list(v) if isinstance(v, tuple) else v for k, v in subst.items()})


def parse_input(input_str):
    """Parses a string input into a structure that can be processed by the unification algorithm."""
    # Remove spaces and handle parentheses
    input_str = input_str.replace(" ", "")

    # Handle compound terms (like p(x, f(y)) -> ['p', 'x', ['f', 'y']])
    def parse_term(term):
        # Handle the compound term
        if '(' in term:
            match = re.match(r'([a-zA-Z0-9_]+)\((.*)\)', term)
            if match:
                predicate = match.group(1)
                arguments_str = match.group(2)
                arguments = [parse_term(arg.strip()) for arg in arguments_str.split(',')]
```

```
            return [predicate] + arguments
        return term

    return parse_term(input_str)


# Main function to interact with the user
def main():
    while True:
        # Get the first and second terms from the user
        expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")
        expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")

        # Parse the input strings into the appropriate structures
        expr1 = parse_input(expr1_input)
        expr2 = parse_input(expr2_input)

        # Perform unification
        is_unified, result = unify_and_check(expr1, expr2)

        # Display the results
        display_result(expr1, expr2, is_unified, result)

        # Ask the user if they want to run another test
        another_test = input("Do you want to test another pair of expressions? (yes/no): ").strip().lower()
        if another_test != 'yes':
            break


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

Output :

```
Enter the first expression (e.g., p(x, f(y))): p(b,x,f(g(z)))
Enter the second expression (e.g., p(a, f(z))): p(z,f(y),f(y))
Expression 1: ['p', 'b', 'x', ['f', ['g', 'z']]]
Expression 2: ['p', 'z', ['f', 'y'], ['f', 'y']]
Result: Unification Successful
Substitutions: {'b': 'z', 'x': ['f', 'y'], 'y': ['g', 'z']}
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(x,h(y))
Enter the second expression (e.g., p(a, f(z))): p(a,f(z))
Expression 1: ['p', 'x', ['h', 'y']]
Expression 2: ['p', 'a', ['f', 'z']]
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(f(a),g(y))
Enter the second expression (e.g., p(a, f(z))): p(x,x)
Expression 1: ['p', ['f', 'a'], ['g', 'y']]
Expression 2: ['p', 'x', 'x']
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): no
```

Program 8

Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:

Lab 8

Forward reasoning Algorithm

Algorithm

function FOL-FC-ASK(KB, α) returns a substitution or false

inputs: KB, the knowledge base, a set of first-order definite clauses, α, the query, an atomic sentence

local variable: new, the sentences inferred on each iteration

repeat until: new is empty
    new ← { }
    for each rule in KB do
        ($p_1 \wedge \dots \wedge p_n \Rightarrow q$) ← Standardize-variables (rule)
        for each θ such that
            SUBST (θ, $p_1 \wedge \dots p_n$) = SUBST (θ, $p_1' \wedge \dots p_n'$)
                for some $p_1' \dots p_n'$ in KB
        q' ← SUBST (θ, q)
        if q' does not unify with some sentence already in KB or new
        Then

Sample:

KB = { facts

American (Robert) : True,

Missile (T1) : True,

Enemy (A, America) : True

Owns (A, T1) : True

Hostile (A) : False

Weapon (T1) : False

Sells (Robert, T1, A) : False

Criminal (Robert) : False

}

Rules = i) American (Robert) and weapon (T1) and sells (Robert, T1, A) and Hostile (A), Criminal (Robert)

ii) Missile (T1) and Owns (A, T1), Sells (Robert, T1, A)

iii) Enemy (A, America), Hostile (A)

O/P

Criminal (Robert) is True

26/11/24

Code:
```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()   # Set of known facts
        self.rules = []      # List of rules

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def infer(self):
        inferred = True
        while inferred:
            inferred = False
            for rule in self.rules:
                if rule.apply(self.facts):
                    inferred = True

# Define the Rule class
class Rule:
    def __init__(self, premises, conclusion):
        self.premises = premises  # List of conditions
        self.conclusion = conclusion  # Conclusion to add if premises are met

    def apply(self, facts):
        if all(premise in facts for premise in self.premises):
            if self.conclusion not in facts:
                facts.add(self.conclusion)
                print(f"Inferred: {self.conclusion}")
                return True
        return False

# Initialize the knowledge base
kb = KnowledgeBase()

# Facts in the problem
kb.add_fact("American(Robert)")
kb.add_fact("Missile(T1)")
kb.add_fact("Owns(A, T1)")
kb.add_fact("Enemy(A, America)")

# Rules based on the problem
# 1. Missile(x) implies Weapon(x)
kb.add_rule(Rule(["Missile(T1)"], "Weapon(T1)"))
```

```
# 2. Enemy(x, America) implies Hostile(x)
kb.add_rule(Rule(["Enemy(A, America)"], "Hostile(A)"))

# 3. Missile(x) and Owns(A, x) imply Sells(Robert, x, A)
kb.add_rule(Rule(["Missile(T1)", "Owns(A, T1)"], "Sells(Robert, T1, A)"))

# 4. American(p) and Weapon(q) and Sells(p, q, r) and Hostile(r) imply Criminal(p)
kb.add_rule(Rule(["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)", "Hostile(A)"],
"Criminal(Robert)"))

# Infer new facts based on the rules
kb.infer()

# Check if Robert is a criminal
if "Criminal(Robert)" in kb.facts:
    print("Conclusion: Robert is a criminal.")
else:
    print("Conclusion: Unable to prove Robert is a criminal.")
```

Output :

```
Inferred: Weapon(T1)
Inferred: Hostile(A)
Inferred: Sells(Robert, T1, A)
Inferred: Criminal(Robert)
Conclusion: Robert is a criminal.
```

Program 9

Create a knowledge base consisting of first order logic statements and prove the given query using Resolution

Algorithm:

~~Lab Typing~~

Lab 9

Convert a given FOL statement into resolution.

Steps

1 Convert all sentences to CNF
2 Negate conclusion & convert result to CNF
3 Add negated conclusion S to the premise clauses
4 Repeat until contradiction or no progress is made:

a) Select 2 clauses & call them parent clauses
b) Resolve them together, performing all required unification
c) If ~~the~~ resolvent is the empty clause, a contradiction has been found (i.e., S follows the premises)
d) If not, add resolvent to the premises

Date __/__/__

Example

a) John likes all kind of food ⇒ $\forall x : food(x) \rightarrow likes(John, x)$

b) Apple & vegetable are food ⇒ $food(apple) \wedge food(vegetable)$

c) Anything anyone eats & not killed is food ⇒ $\forall x \forall y \; Eats(x, y) \wedge \neg Killed(x) \rightarrow food(y)$

d) Anil eats peanuts & still alive ⇒ $Eats(anil, peanuts) \wedge Alive(anil)$

e) Harry eats everything that anil eats ⇒ $\forall x : Eats(anil, x) \rightarrow Eats(harry, x)$

f) Anyone who is alive implies not killed ⇒ $\forall x : \neg killed(x) \rightarrow alive(x)$

g) Anyone who is not killed implies alive ⇒ $\forall x : Alive(x) \rightarrow \neg killed(x)$

Prove by resolution ⇒ h) John likes peanuts.

result ⇒ Does John like peanuts? yes, proven by resolution

Code:

```
from sympy import symbols, And, Or, Not, Implies, to_cnf

# Define constants (entities in the problem)
John, Anil, Harry, Apple, Vegetables, Peanuts, x, y = symbols('John Anil Harry Apple Vegetables
Peanuts x y')

# Define predicates as symbols (this works as a workaround)
Food = symbols('Food')
Eats = symbols('Eats')
Likes = symbols('Likes')
Alive = symbols('Alive')
Killed = symbols('Killed')

# Knowledge Base (Premises) in First-Order Logic
premises = [
    # 1. John likes all kinds of food: Food(x) → Likes(John, x)
    Implies(Food, Likes),

    # 2. Apples and vegetables are food: Food(Apple) ∧ Food(Vegetables)
    And(Food, Food),

    # 3. Anything anyone eats and is not killed is food: (Eats(y, x) ∧ ¬Killed(y)) → Food(x)
    Implies(And(Eats, Not(Killed)), Food),

    # 4. Anil eats peanuts and is still alive: Eats(Anil, Peanuts) ∧ Alive(Anil)
    And(Eats, Alive),

    # 5. Harry eats everything that Anil eats: Eats(Anil, x) → Eats(Harry, x)
    Implies(Eats, Eats),

    # 6. Anyone who is alive implies not killed: Alive(x) → ¬Killed(x)
    Implies(Alive, Not(Killed)),

    # 7. Anyone who is not killed implies alive: ¬Killed(x) → Alive(x)
    Implies(Not(Killed), Alive),
]

# Negated conclusion to prove: ¬Likes(John, Peanuts)
negated_conclusion = Not(Likes)

# Convert all premises and the negated conclusion to Conjunctive Normal Form (CNF)
cnf_clauses = [to_cnf(premise, simplify=True) for premise in premises]
cnf_clauses.append(to_cnf(negated_conclusion, simplify=True))

# Function to resolve two clauses
```

```
def resolve(clause1, clause2):
    """
    Resolve two CNF clauses to produce resolvents.
    """
    clause1_literals = clause1.args if isinstance(clause1, Or) else [clause1]
    clause2_literals = clause2.args if isinstance(clause2, Or) else [clause2]
    resolvents = []

    for literal in clause1_literals:
        if Not(literal) in clause2_literals:
            # Remove the literal and its negation and combine the rest
            new_clause = Or(
                *[l for l in clause1_literals if l != literal],
                *[l for l in clause2_literals if l != Not(literal)]
            ).simplify()
            resolvents.append(new_clause)

    return resolvents

# Function to perform resolution on the set of CNF clauses
def resolution(cnf_clauses):
    """
    Perform resolution on CNF clauses to check for a contradiction.
    """
    clauses = set(cnf_clauses)
    new_clauses = set()

    while True:
        clause_list = list(clauses)

        for i in range(len(clause_list)):
            for j in range(i + 1, len(clause_list)):
                resolvents = resolve(clause_list[i], clause_list[j])
                if False in resolvents:  # Empty clause found
                    return True  # Contradiction found; proof succeeded
                new_clauses.update(resolvents)

        if new_clauses.issubset(clauses):  # No new information
            return False  # No contradiction; proof failed

        clauses.update(new_clauses)

# Perform resolution to check if the conclusion follows
result = resolution(cnf_clauses)
print("Does John like peanuts? ", "Yes, proven by resolution." if result else "No, cannot be proven.")
```

OUTPUT:

```
Does John like peanuts?  Yes, proven by resolution.
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:

Lab 10
Alpha Beta Pruning

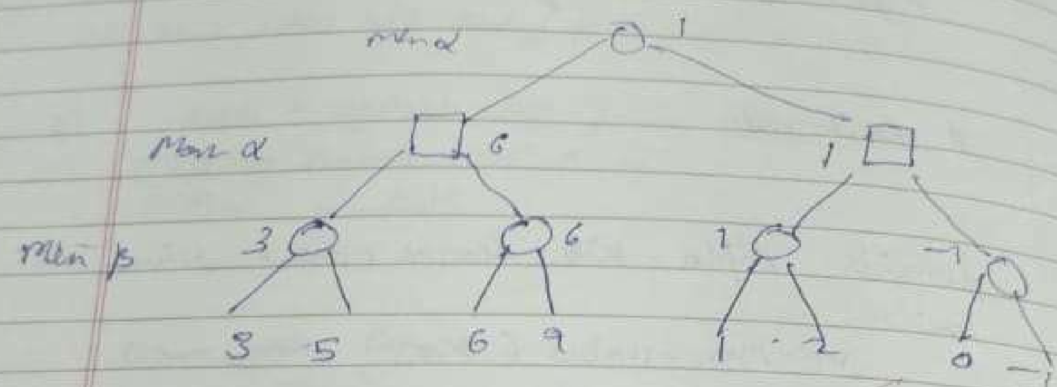
Pseudo code

function alpha-beta Search (state) return an action
    $V \leftarrow$ Max-Value (state, $-\infty$, $+\infty$)
    return the action in Actions (state) with value V

function Max value(state, α, β) return utility value
    If Terminal test (state) then return utility (state)
    $V \leftarrow -\infty$
    for each a in Actions (state) do
        $V \leftarrow$ Max (V, Min-Value (Result (s,a), α, β))
        If $V \geq \beta$ then return V
        $\alpha \leftarrow$ Max (α, V)
    return V

function Min Value (state, α, β) return a utility value
    If Terminal ~~state~~ test (state) then return utility (state)
    $V \leftarrow +\infty$
    for each a in actions (state) do
        $V \leftarrow$ Min (V, Max-value (Result (s,a), α, β))

Example



result => 1

Code:

```
# Python3 program to demonstrate
# working of Alpha-Beta Pruning with detailed step output

# Initial values of Alpha and Beta
MAX, MIN = 1000, -1000

# Returns optimal value for the current player
def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
   # Terminating condition: leaf node is reached
   if depth == 3:
      print(f"Leaf node reached: Depth={depth}, NodeIndex={nodeIndex},
Value={values[nodeIndex]}")
      return values[nodeIndex]

   if maximizingPlayer:
      best = MIN
      print(f"Maximizer: Depth={depth}, NodeIndex={nodeIndex}, Alpha={alpha}, Beta={beta}")

      # Recur for left and right children
      for i in range(2):
         val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
         best = max(best, val)
         alpha = max(alpha, best)
         print(f"Maximizer updated: Depth={depth}, NodeIndex={nodeIndex}, Best={best},
Alpha={alpha}, Beta={beta}")

         # Alpha Beta Pruning
         if beta <= alpha:
            print(f"Maximizer Pruned: Depth={depth}, NodeIndex={nodeIndex}, Alpha={alpha},
Beta={beta}")
            break
      return best
   else:
      best = MAX
      print(f"Minimizer: Depth={depth}, NodeIndex={nodeIndex}, Alpha={alpha}, Beta={beta}")

      # Recur for left and right children
      for i in range(2):
         val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
         best = min(best, val)
         beta = min(beta, best)
         print(f"Minimizer updated: Depth={depth}, NodeIndex={nodeIndex}, Best={best},
Alpha={alpha}, Beta={beta}")

         # Alpha Beta Pruning
```

```
            if beta <= alpha:
                print(f"Minimizer Pruned: Depth={depth}, NodeIndex={nodeIndex}, Alpha={alpha}, Beta={beta}")
                break
        return best

# Driver Code
if __name__ == "__main__":
    values = [3, 5, 6, 9, 1, 2, 0, -1]  # Leaf node values
    print("Starting Alpha-Beta Pruning...")
    optimal_value = minimax(0, 0, True, values, MIN, MAX)
    print(f"\nThe optimal value is: {optimal_value}")
```

OUTPUT:

```
Starting Alpha-Beta Pruning...
Maximizer: Depth=0, NodeIndex=0, Alpha=-1000, Beta=1000
Minimizer: Depth=1, NodeIndex=0, Alpha=-1000, Beta=1000
Maximizer: Depth=2, NodeIndex=0, Alpha=-1000, Beta=1000
Leaf node reached: Depth=3, NodeIndex=0, Value=3
Maximizer updated: Depth=2, NodeIndex=0, Best=3, Alpha=3, Beta=1000
Leaf node reached: Depth=3, NodeIndex=1, Value=5
Maximizer updated: Depth=2, NodeIndex=0, Best=5, Alpha=5, Beta=1000
Minimizer updated: Depth=1, NodeIndex=0, Best=5, Alpha=-1000, Beta=5
Maximizer: Depth=2, NodeIndex=1, Alpha=-1000, Beta=5
Leaf node reached: Depth=3, NodeIndex=2, Value=6
Maximizer updated: Depth=2, NodeIndex=1, Best=6, Alpha=6, Beta=5
Maximizer Pruned: Depth=2, NodeIndex=1, Alpha=6, Beta=5
Minimizer updated: Depth=1, NodeIndex=0, Best=5, Alpha=-1000, Beta=5
Maximizer updated: Depth=0, NodeIndex=0, Best=5, Alpha=5, Beta=1000
Minimizer: Depth=1, NodeIndex=1, Alpha=5, Beta=1000
Maximizer: Depth=2, NodeIndex=2, Alpha=5, Beta=1000
Leaf node reached: Depth=3, NodeIndex=4, Value=1
Maximizer updated: Depth=2, NodeIndex=2, Best=1, Alpha=5, Beta=1000
Leaf node reached: Depth=3, NodeIndex=5, Value=2
Maximizer updated: Depth=2, NodeIndex=2, Best=2, Alpha=5, Beta=1000
Minimizer updated: Depth=1, NodeIndex=1, Best=2, Alpha=5, Beta=2
Minimizer Pruned: Depth=1, NodeIndex=1, Alpha=5, Beta=2
Maximizer updated: Depth=0, NodeIndex=0, Best=5, Alpha=5, Beta=1000

The optimal value is: 5
```