

**VISVESVARAYA TECHNOLOGICAL  
UNIVERSITY**

“JnanaSangama”, Belgaum -590014, Karnataka.



**LAB REPORT  
on**

**Artificial Intelligence (23CS5PCAIN)**

*Submitted by*

**Shivaraj Kallappa Pujari (1BM22CS259)**

*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 **Shivaraj Kallappa Pujari (1BM22CS259)**, 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.

Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE
--	---

## Index

<b>Sl. No.</b>	<b>Date</b>	<b>Experiment Title</b>	<b>Page No.</b>
1	30-9-2024	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	1-7
2	7-10-2024	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	7-16
3	14-10-2024	Implement A* search algorithm	17-26
4	21-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	27-33
5	28-10-2024	Simulated Annealing to Solve 8-Queens problem	34-36
6	11-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	37-39
7	2-12-2024	Implement unification in first order logic	40-44
8	2-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	44-48
9	16-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	48-52
10	16-12-2024	Implement Alpha-Beta Pruning.	52-57

Github Link:

[https://github.com/Shivarajpujari2004/Ai\\_Lab](https://github.com/Shivarajpujari2004/Ai_Lab)

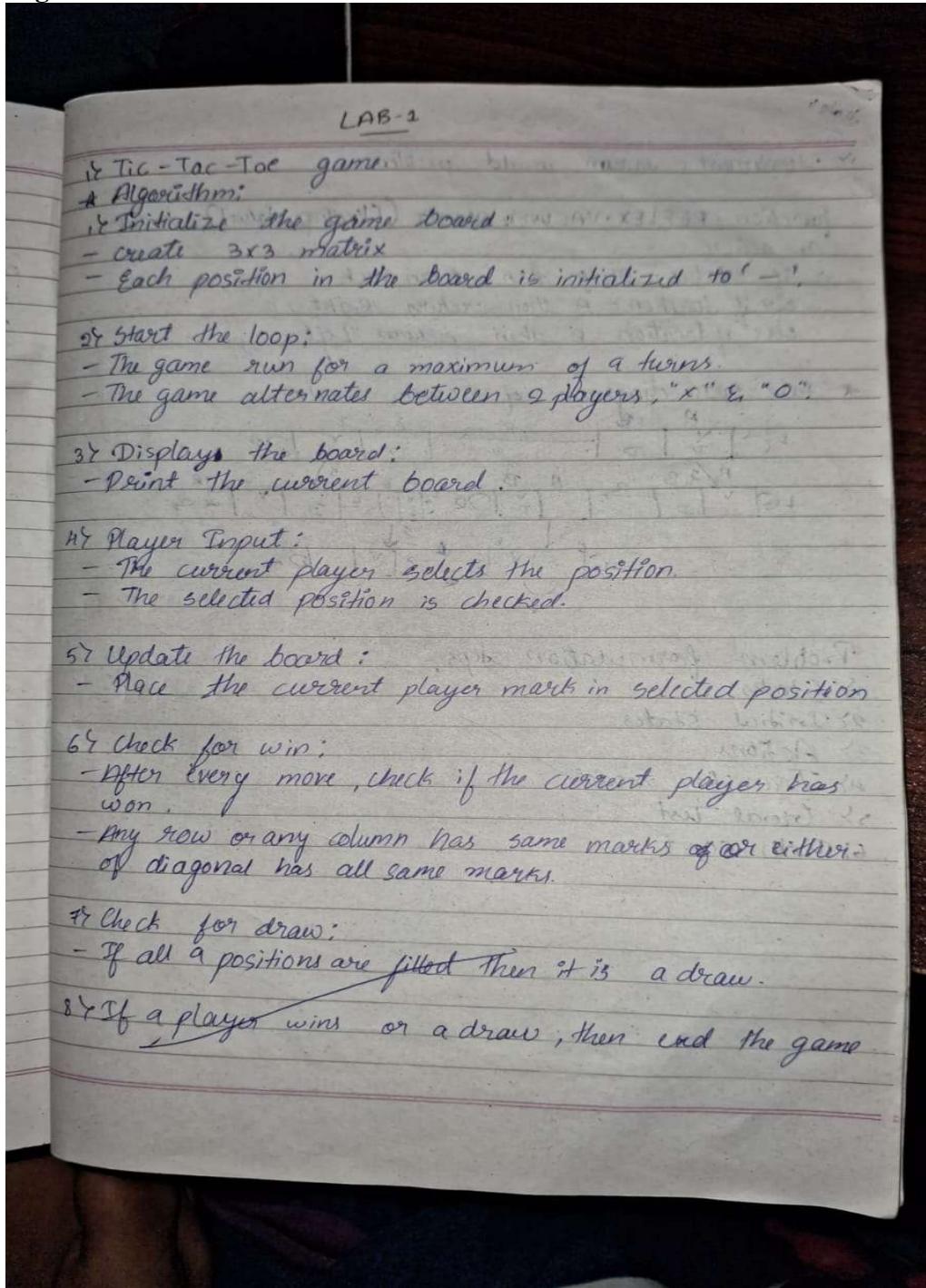
## Program 1

Implement Tic - Tac - Toe Game

Implement vacuum cleaner agent

### 1: Tic - Tac - Toe

#### Algorithm:



#### Code:

```

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:

WIN:

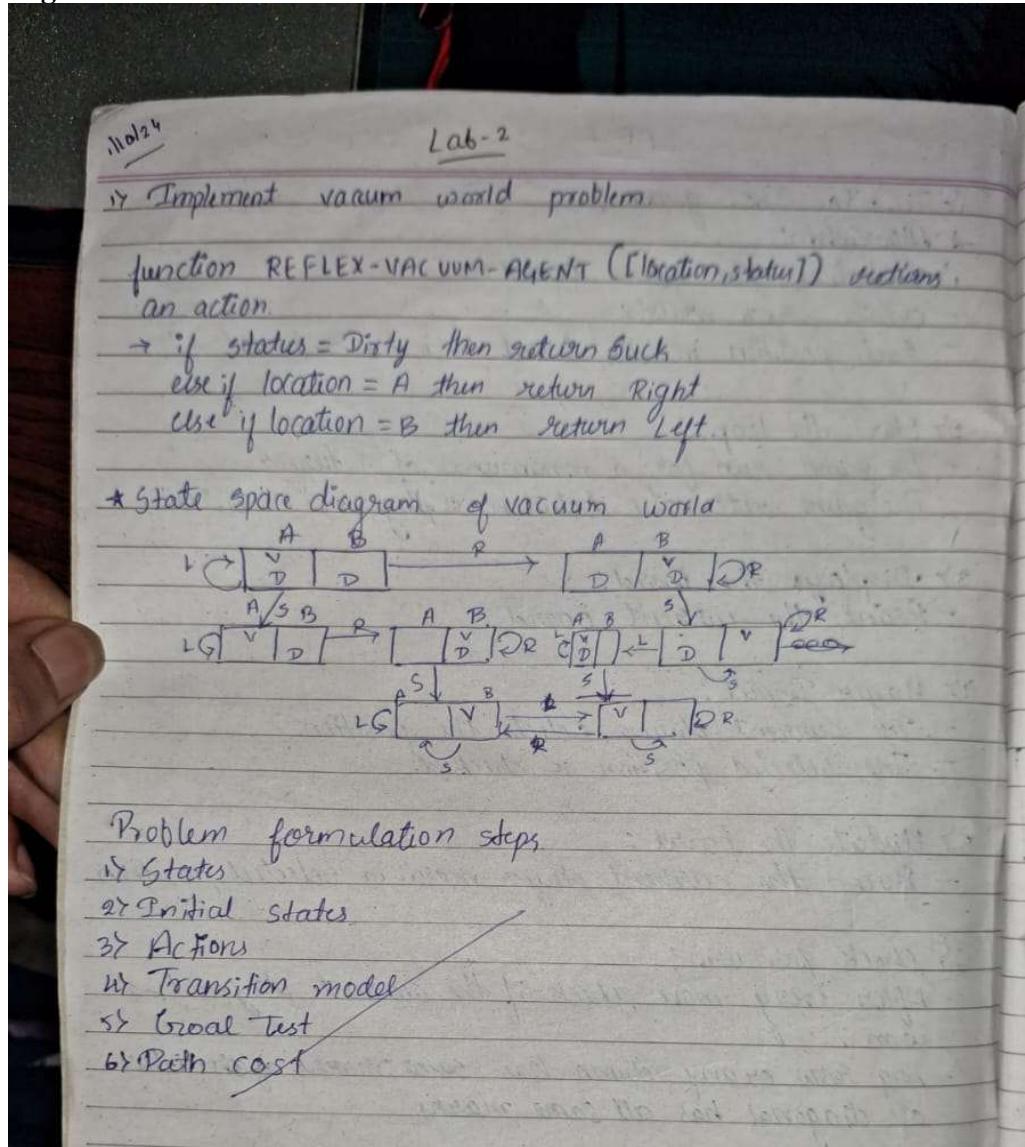
```
- | - | -
- | - | -
- | - | -
User1's turn (mark: X)
Enter the row (0-2): 0
Enter the column (0-2): 0
X | - | -
- | - | -
- | - | -
User2's turn (mark: O)
Enter the row (0-2): 0
Enter the column (0-2): 1
X | O | -
- | - | -
- | - | -
User1's turn (mark: X)
Enter the row (0-2): 0
Enter the column (0-2): 2
X | O | X
- | - | -
- | - | -
User2's turn (mark: O)
Enter the row (0-2): 1
Enter the column (0-2): 1
X | O | X
- | O | -
- | - | -
User1's turn (mark: X)
Enter the row (0-2): 1
Enter the column (0-2): 0
X | O | X
- | O | -
- | - | -
User2's turn (mark: O)
Enter the row (0-2): 2
Enter the column (0-2): 1
X | O | X
- | O | -
- | O | -
User2wins!
```

## DRAW:

```
- | - | -
-----+
- | - | -
-----+
- | - | -
-----+
User1's turn (mark: X)
Enter the row (0-2): 0
Enter the column (0-2): 1
- | X | -
-----+
- | - | -
-----+
- | - | -
-----+
User2's turn (mark: O)
Enter the row (0-2): 0
Enter the column (0-2): 0
O | X | -
-----+
- | - | -
-----+
- | - | -
-----+
User1's turn (mark: X)
Enter the row (0-2): 1
Enter the column (0-2): 1
O | X | -
-----+
- | X | -
-----+
- | - | -
-----+
User2's turn (mark: O)
Enter the row (0-2): 2
Enter the column (0-2): 1
O | X | -
-----+
- | X | -
-----+
- | O | -
-----+
User1's turn (mark: X)
Enter the row (0-2): 2
Enter the column (0-2): 0
O | X | -
-----+
- | X | -
-----+
X | O | -
-----+
User2's turn (mark: O)
Enter the row (0-2): 0
Enter the column (0-2): 2
O | X | 0
-----+
- | X | -
-----+
X | O | -
-----+
User1's turn (mark: X)
Enter the row (0-2): 2
Enter the column (0-2): 2
O | X | 0
-----+
- | X | -
-----+
X | O | X
-----+
User2's turn (mark: O)
Enter the row (0-2): 1
Enter the column (0-2): 1
Cell already taken. Try again.
O | X | 0
-----+
- | X | -
-----+
X | O | X
-----+
User1's turn (mark: X)
Enter the row (0-2): 1
Enter the column (0-2): 2
O | X | 0
-----+
- | X | -
-----+
X | O | X
-----+
It's a draw!
```

## 2. Vacuum Cleaner :

### Algorithm:



A Algorithm:-

1) Initialize variables:

→ Create an empty list room & set cost=0

2) Check the rooms are clean or not by checking status.

→ If room's status is dirty then clean it & move to other room.

3) Initialize the goal status : {A:0, B:0}

→ If room status = goal status  
then all rooms are cleaned

Q8

11/10/24

#### Code:

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 for Room A (1 for dirty, 0 for clean): 1
Enter status for Room B (1 for dirty, 0 for clean): 1
Room A cleaned
Current Status: A: 0, B: 1
Moving to room B
Room B cleaned
Current Status: A: 0, B: 0
Moving to room A
All rooms cleaned
Total cost: 2
```

### Program 2

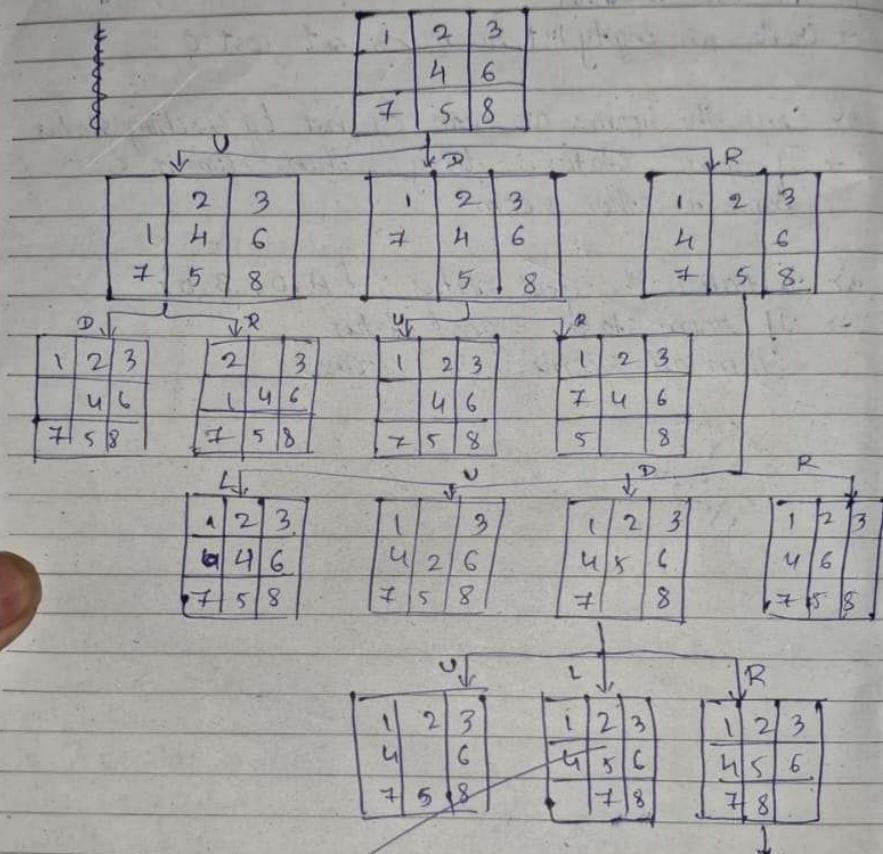
Implement 8 puzzle problems using Depth First Search (DFS)  
Implement Iterative deepening search algorithm

### Algorithm:

8/10/24

### Lab-3

\* Implement an algorithm to solve 8 Puzzle problem:-



\* Algorithm:- BFS

- It Initialize: Define the function that accepts the initial state
  - Define goal state
  - Initialize a queue using a deque to hold states starting with an empty path.
  - Create a set to track visited states to avoid revisiting.

3) Find the Blank state:-

→ Define a helper function to locate the position of blank tile.

3) Generating possible moves:

→ Determine the position of blank tile.

→ Check if the move is valid, if valid, create a new state by swapping blank tile with adjacent tile.

4) BFS Loop:

→ While the queue is not empty.

→ Dequeue the current state & its path.

→ Check if the current state is goal state, return.

→ Generate all moves.

→ For next state, if state has not been visited, enqueue the new state along with updated path.

→ mark new state as visited.

5) Termination

→ If solution is found, print each step along with total counts.

SFS  
81/1024

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 L move Current state: 2 3 6 1 4 8 0 7 5	Took a L move Current state: 1 0 2 4 6 3 7 5 8
Took a U move Current state: 0 2 3 1 4 6 7 5 8	Took a L move Current state: 2 3 6 1 0 4 7 5 8	Took a L move Current state: 0 1 2 4 6 3 7 5 8
Took a R move Current state: 2 0 3 1 4 6 7 5 8	Took a D move Current state: 2 3 6 1 5 4 7 0 8	Took a D move Current state: 1 2 3 4 6 8 7 5 0
Took a R move Current state: 2 3 0 1 4 6 7 5 8	Took a R move Current state: 2 3 6 1 5 4 7 8 0	Took a L move Current state: 1 2 3 4 6 8 7 0 5
Took a D move Current state: 2 3 6 1 4 0 7 5 8	Took a L move Current state: 2 3 6 1 5 4 0 7 8	Took a L move Current state: 1 2 3 4 6 8 0 7 5
Took a D move Current state: 2 3 6 1 4 8 7 5 0	Took a D move Current state: 2 4 3 1 0 6 7 5 8	Took a D move Current state: 1 2 3 4 5 6 7 0 8
Took a L move Current state: 2 3 6 1 4 8 7 0 5	Took a R move Current state: 2 4 3 1 6 0 7 5 8	Took a R move Current state: 1 2 3 4 5 6 7 8 0
Took a U move Current state: 2 3 6 1 0 8 7 4 5	Took a U move Current state: 2 4 0 1 6 3 7 5 8	Number of states : 42 Solution found: True

## 2 : Iterative deepening search

Algorithm:

Lab - 4

depth first search

Qn't Solving 8 Puzzle problem using iterative deepening approach.

\* Algorithm:

- 1) For each child of current node.
- 2) If it is target node, return.
- 3) If the current maximum depth is reached, return.
- 4) Set the current node to this node & go back to 1.
- 5) After having gone through all children, go to the next child of parent (the next sibling).
- 6) After having gone through all children of start node, increase the max depth & go back to 1.
- 7) If we have reached at leaf (bottom) nodes, the goal node doesn't exist.

function Iterative-deepening-search (problem) returns a solution or failure

```
for depth = 0 to  $\infty$  do
    res  $\leftarrow$  depth-limited-search (problem, depth)
    if (res  $\neq$  cutoff) then return result
```

Initial

1	2	3
4		6
7	5	8

1	2	3
4	5	6
7	8	-

Depth = 0

	1	2	3	
4		6		
7	5	8		

R	D	L	R
1 3	1 2 3	1 2 3	1 2 3
4 2 6	4 5 6	4 6	4 6
7 5 8	7 8	7 8	7 8

R	L	R
1 3	1 3	1 2 3
4 2 6	4 2 6	4 5 6
7 5 8	7 5 8	7 8

Goal

found at depth 2

22 total

8 N Queen state space tree

(1, 2)	(1, 3)	(1, 4)	(2, 3)	(3, 4)
4	4	4	4	4
4	4	4	4	4
4	4	4	4	4

(2, 4)	(2, 4)	(2, 4)	(2, 4)
4	4	4	4
4	4	4	4

## Code:

```
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]
[7, 4, 6]
[0, 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']

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

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']

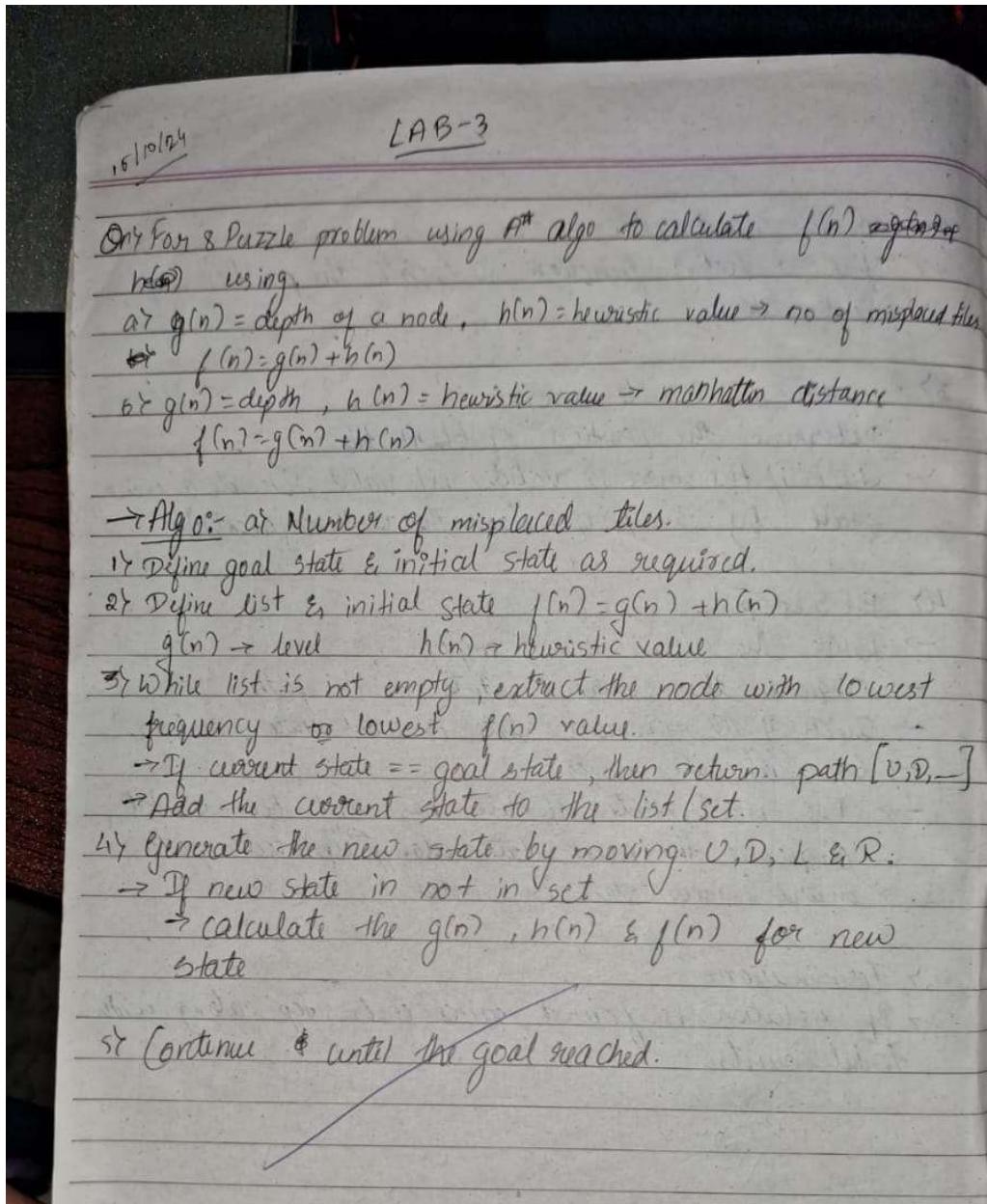
Current state: [1, 2, 3]
[4, 6, 0]
[7, 5, 8]
Moves: ['Right', 'Right']

Goal not found within max depth.
```

### Program 3

Implement A\* search algorithm

**Algorithm:**



only draw the state space diagram for

2	8	3
1	6	4
7	5	

Goal  $\Rightarrow$

1	2	3
8	4	
7	6	5

Find the most effective path using misplaced tiles.

2	8	3
1	6	4
7	5	

$$g = 0$$

$$h = 4$$

L	U	R																												
<table border="1"> <tr> <td>2</td><td>8</td><td>3</td></tr> <tr> <td>1</td><td>6</td><td>4</td></tr> <tr> <td>7</td><td>5</td><td></td></tr> </table>	2	8	3	1	6	4	7	5		<table border="1"> <tr> <td>2</td><td>8</td><td>3</td></tr> <tr> <td>1</td><td>4</td><td></td></tr> <tr> <td>7</td><td>6</td><td>5</td></tr> </table>	2	8	3	1	4		7	6	5	<table border="1"> <tr> <td>2</td><td>8</td><td>3</td></tr> <tr> <td>1</td><td>6</td><td>4</td></tr> <tr> <td>7</td><td>5</td><td></td></tr> </table>	2	8	3	1	6	4	7	5		$g(n) = 1$
2	8	3																												
1	6	4																												
7	5																													
2	8	3																												
1	4																													
7	6	5																												
2	8	3																												
1	6	4																												
7	5																													
$h(n) = 5$	$h(n) = 3$	$h(n) = 5$																												

L	U	P	R																																					
<table border="1"> <tr> <td>2</td><td>3</td><td></td></tr> <tr> <td>1</td><td>8</td><td>4</td></tr> <tr> <td>7</td><td>6</td><td>5</td></tr> </table>	2	3		1	8	4	7	6	5	<table border="1"> <tr> <td>2</td><td>8</td><td>3</td></tr> <tr> <td>1</td><td>4</td><td></td></tr> <tr> <td>7</td><td>6</td><td>5</td></tr> </table>	2	8	3	1	4		7	6	5	<table border="1"> <tr> <td>2</td><td>8</td><td>3</td></tr> <tr> <td>1</td><td>6</td><td>4</td></tr> <tr> <td>7</td><td>5</td><td></td></tr> </table>	2	8	3	1	6	4	7	5		<table border="1"> <tr> <td>2</td><td>8</td><td>3</td></tr> <tr> <td>1</td><td>4</td><td></td></tr> <tr> <td>7</td><td>6</td><td>5</td></tr> </table>	2	8	3	1	4		7	6	5	$g(n) = 2$
2	3																																							
1	8	4																																						
7	6	5																																						
2	8	3																																						
1	4																																							
7	6	5																																						
2	8	3																																						
1	6	4																																						
7	5																																							
2	8	3																																						
1	4																																							
7	6	5																																						
$h(n) = 3$	$h(n) = 3$	$+ h(n) = 4$	$h(n) = 5$																																					

L	R	U	D																																					
<table border="1"> <tr> <td>2</td><td>3</td><td></td></tr> <tr> <td>1</td><td>8</td><td>4</td></tr> <tr> <td>7</td><td>6</td><td>5</td></tr> </table>	2	3		1	8	4	7	6	5	<table border="1"> <tr> <td>2</td><td>3</td><td></td></tr> <tr> <td>1</td><td>8</td><td>4</td></tr> <tr> <td>7</td><td>6</td><td>5</td></tr> </table>	2	3		1	8	4	7	6	5	<table border="1"> <tr> <td>8</td><td>3</td><td></td></tr> <tr> <td>2</td><td>1</td><td>4</td></tr> <tr> <td>7</td><td>6</td><td>5</td></tr> </table>	8	3		2	1	4	7	6	5	<table border="1"> <tr> <td>2</td><td>8</td><td>3</td></tr> <tr> <td>7</td><td>1</td><td>4</td></tr> <tr> <td>6</td><td>5</td><td></td></tr> </table>	2	8	3	7	1	4	6	5		$g(n) = 3$
2	3																																							
1	8	4																																						
7	6	5																																						
2	3																																							
1	8	4																																						
7	6	5																																						
8	3																																							
2	1	4																																						
7	6	5																																						
2	8	3																																						
7	1	4																																						
6	5																																							
$h(n) = 2$	$h(n) = 4$	$h(n) = 3$	$h(n) = 4$																																					

D	R																													
<table border="1"> <tr> <td>1</td><td>2</td><td>3</td></tr> <tr> <td>8</td><td>4</td><td></td></tr> <tr> <td>7</td><td>6</td><td>5</td></tr> </table>	1	2	3	8	4		7	6	5	<table border="1"> <tr> <td>2</td><td>3</td><td></td></tr> <tr> <td>1</td><td>8</td><td>4</td></tr> <tr> <td>7</td><td>6</td><td>5</td></tr> </table>	2	3		1	8	4	7	6	5	<table border="1"> <tr> <td>8</td><td>3</td><td></td></tr> <tr> <td>2</td><td>1</td><td>4</td></tr> <tr> <td>7</td><td>6</td><td>5</td></tr> </table>	8	3		2	1	4	7	6	5	$g(n) = 4$
1	2	3																												
8	4																													
7	6	5																												
2	3																													
1	8	4																												
7	6	5																												
8	3																													
2	1	4																												
7	6	5																												
$h(n) = 1$																														

U	R	D																												
<table border="1"> <tr> <td>1</td><td>2</td><td>3</td></tr> <tr> <td>8</td><td>4</td><td></td></tr> <tr> <td>7</td><td>6</td><td>5</td></tr> </table>	1	2	3	8	4		7	6	5	<table border="1"> <tr> <td>2</td><td>3</td><td></td></tr> <tr> <td>1</td><td>8</td><td>4</td></tr> <tr> <td>7</td><td>6</td><td>5</td></tr> </table>	2	3		1	8	4	7	6	5	<table border="1"> <tr> <td>1</td><td>2</td><td>3</td></tr> <tr> <td>7</td><td>8</td><td>4</td></tr> <tr> <td>6</td><td>5</td><td></td></tr> </table>	1	2	3	7	8	4	6	5		$g(n) = 5$
1	2	3																												
8	4																													
7	6	5																												
2	3																													
1	8	4																												
7	6	5																												
1	2	3																												
7	8	4																												
6	5																													
$h(n) = 1$																														

$$\Rightarrow \text{Cost} \Rightarrow f(n) = g(n) + h(n) = 5$$

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

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

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

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

```

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]

```

```

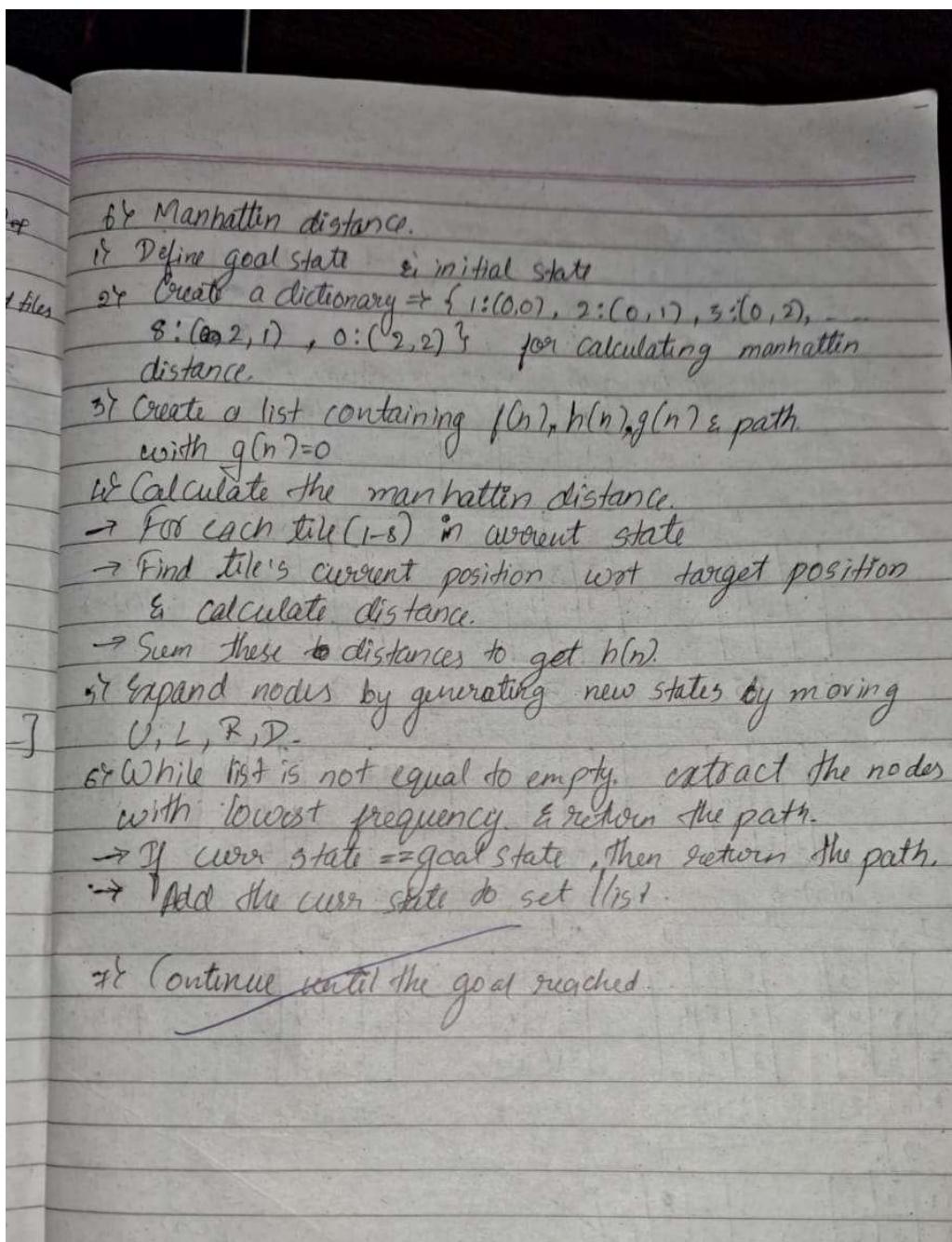
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

```

**Code :**

### **Manhattan distance approach**

**Algorithm:**



$$\begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 2 = 6 \end{array}$$

Q) Draw the state space diagram using manhattan distance

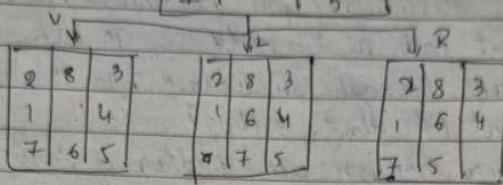
Initial

2	8	3
1	6	4
7	5	1

Goal : 1 2 3

8 4

7 6 5

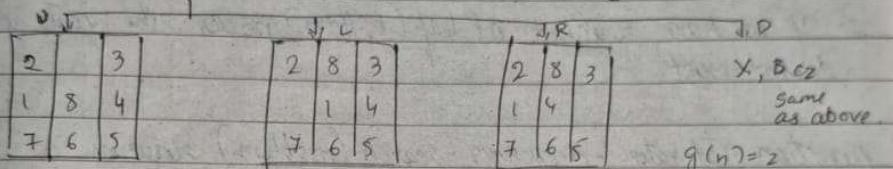


$g(n)=1$

$h(n)=4$

$h(n)=8$

$h(n)=6$



$x, B.C_2$

same as above.

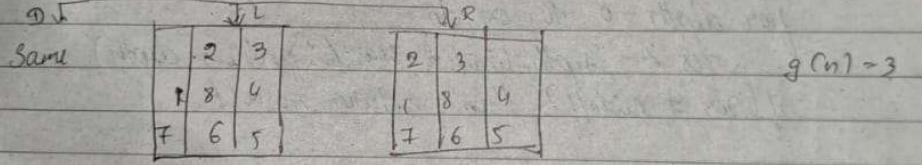
$g(n)=2$

$n=2$

$h(n)=3$

$h(n)=5$

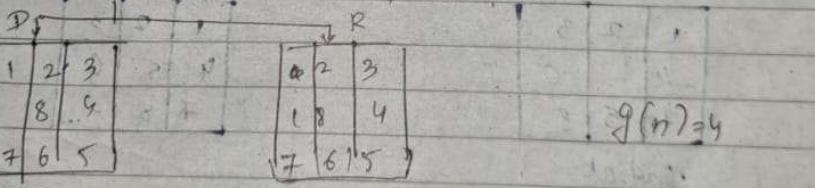
$h(n)=5$



$g(n)=3$

$h(n)=2$

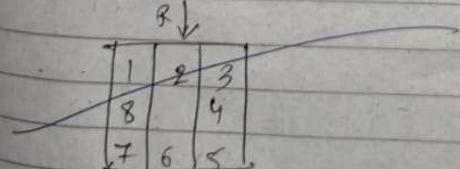
$h(n)=5$



$g(n)=4$

$h(n)=1$

$h(n)=3$



$g(n)=5$

~~h(n)=0~~

$f(n) = g(n) + h(n) = 5$

**Code:**

```
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.")

```

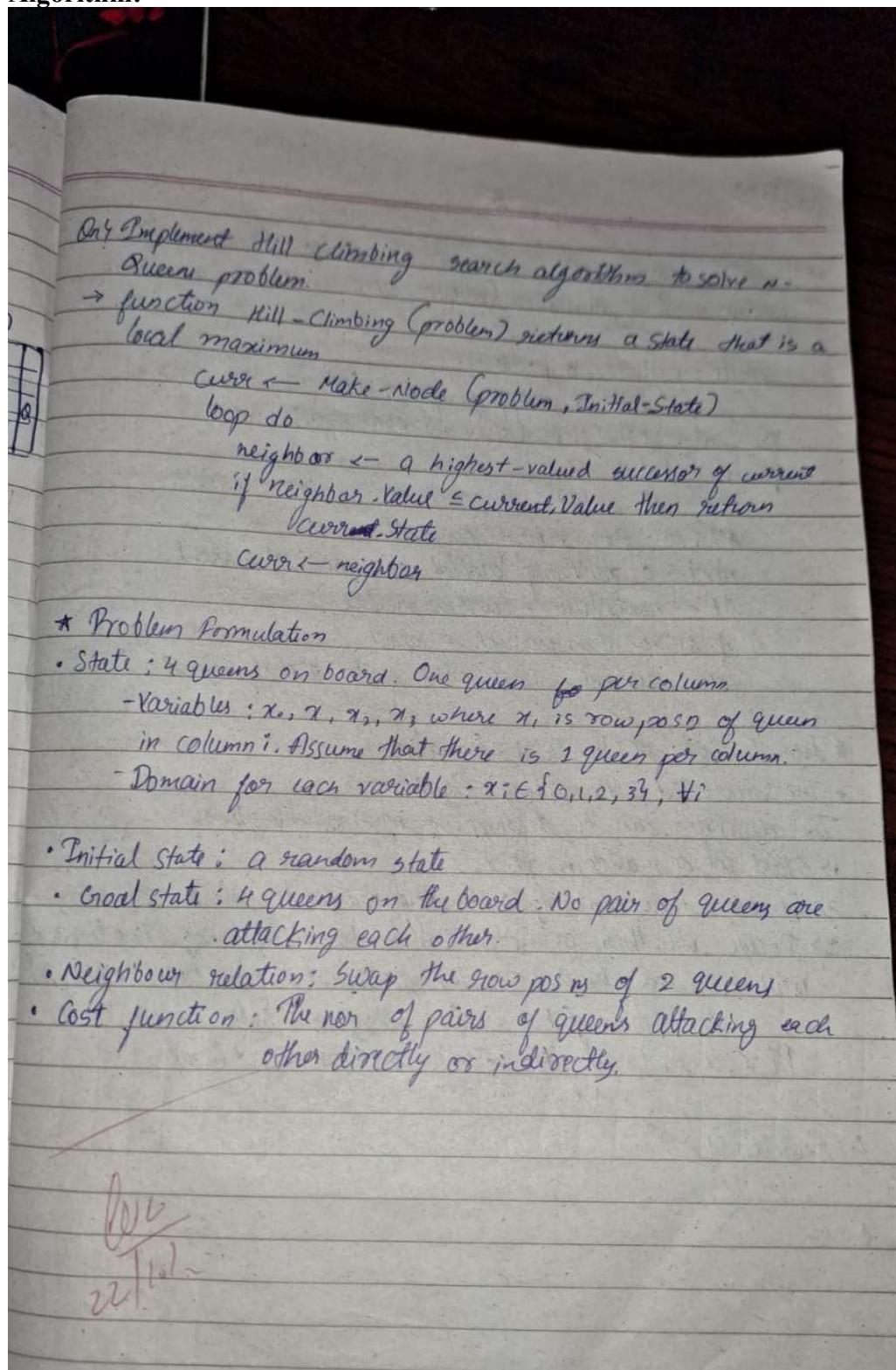
## Output :

<pre> Current state: [2, 8, 3] [1, 6, 4] [7, 0, 5] Move: Start Heuristic value (Manhattan distance): 5 Cost to reach this node: 0 </pre>	<pre> Current state: [1, 2, 3] [8, 0, 4] [7, 6, 5] Move: Left Heuristic value (Manhattan distance): 0 Cost to reach this node: 5 </pre>
<pre> Current state: [2, 8, 3] [1, 0, 4] [7, 6, 5] Move: Down Heuristic value (Manhattan distance): 4 Cost to reach this node: 1 </pre>	<pre> Solution path: [2, 8, 3] [1, 6, 4] [7, 0, 5] </pre>
<pre> Current state: [2, 0, 3] [1, 8, 4] [7, 6, 5] Move: Down Heuristic value (Manhattan distance): 3 Cost to reach this node: 2 </pre>	<pre> [2, 0, 3] [1, 8, 4] [7, 6, 5] </pre>
<pre> Current state: [0, 2, 3] [1, 8, 4] [7, 6, 5] Move: Right Heuristic value (Manhattan distance): 2 Cost to reach this node: 3 </pre>	<pre> [0, 2, 3] [1, 8, 4] [7, 6, 5] </pre>
	<pre> [1, 2, 3] [8, 0, 4] [7, 6, 5] </pre>

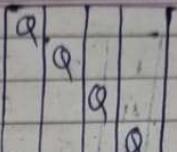
## Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

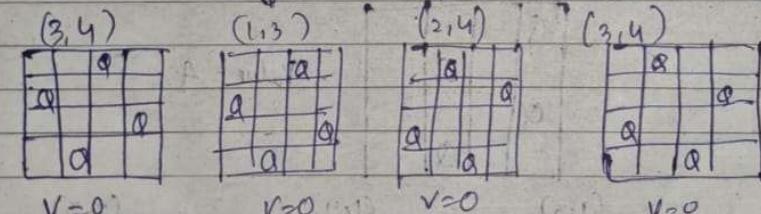
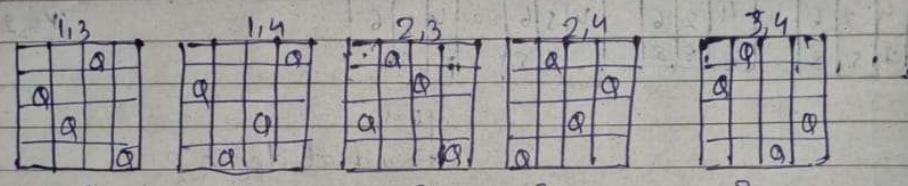
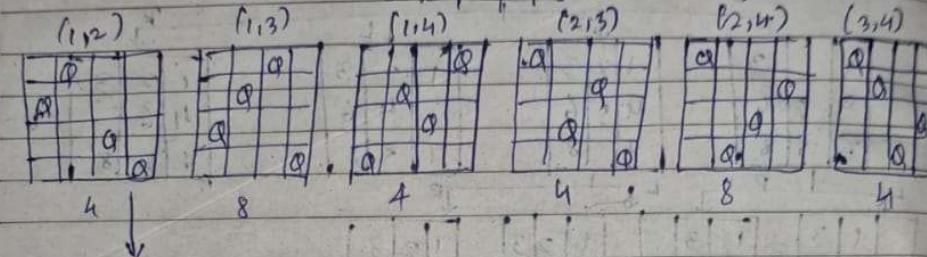
### Algorithm:



\* N Queen State Space Tree.



$V=12$



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

```

```

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  
29/10/2022

\* Write a program to implement Simulated Annealing Algorithm  
→ function simulated-Annealing (problem, schedule) returns a solution state

inputs : problem, a problem.  
schedule, a mapping from time to "temperature"  
current  $\leftarrow$  Make-Node (problem, Initial.state)  
for  $t = 1$  to  $\infty$  do  
     $T \leftarrow$  schedule ( $t$ )  
    if  $T = 0$  then return current  
    next  $\leftarrow$  a randomly selected successor of current  
     $\Delta F \leftarrow$  next.Value - current.Value  
    if  $\Delta F \geq 0$  then current  $\leftarrow$  next  
    else current  $\leftarrow$  next only with probability  $e^{\Delta F/T}$

\* Introduction to simulated Annealing

- The Simulated Annealing Algorithm

The algorithm can be decomposed in 4 simple steps:

- 1) Start at a random pt  $x$ .
- 2) Choose a new pt  $x_j$  on a neighbourhood  $N(x)$
- 3) Decide whether or not to move to new pt  $x_j$ . The decision will be made based on the probability function  $P(x, x_j, T)$
- 4) Reduce  $T$

$$P(x, x_j, T) = \begin{cases} 1 & \text{or } F(x_j) > F(x) \\ e^{\frac{F(x_j) - F(x)}{T}} & \text{if } F(x_j) \leq F(x) \end{cases}$$

1)

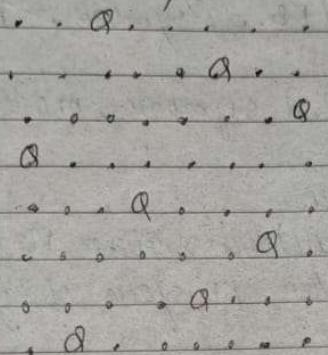
→ Output of 8 queens problem:

→ initial position = [4, 6, 1, 5, 2, 0, 3, 7]

→ Output :-

→ The best position found is: [2 5 7 0 3 6 4 ]  
The no. of queens that are not attacking each other: 8.0

Board Representation:



2) Tower of Hanoi:

→ Output:

Best state: [2 2 2]

No. of correct disks on destination peg: 3.0

Tower of Hanoi configuration:

Peg 0: [ ]

Peg 1: [ ]

Peg 2: [0, 1, 2]

BB  
2^10/2^n

**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)
```

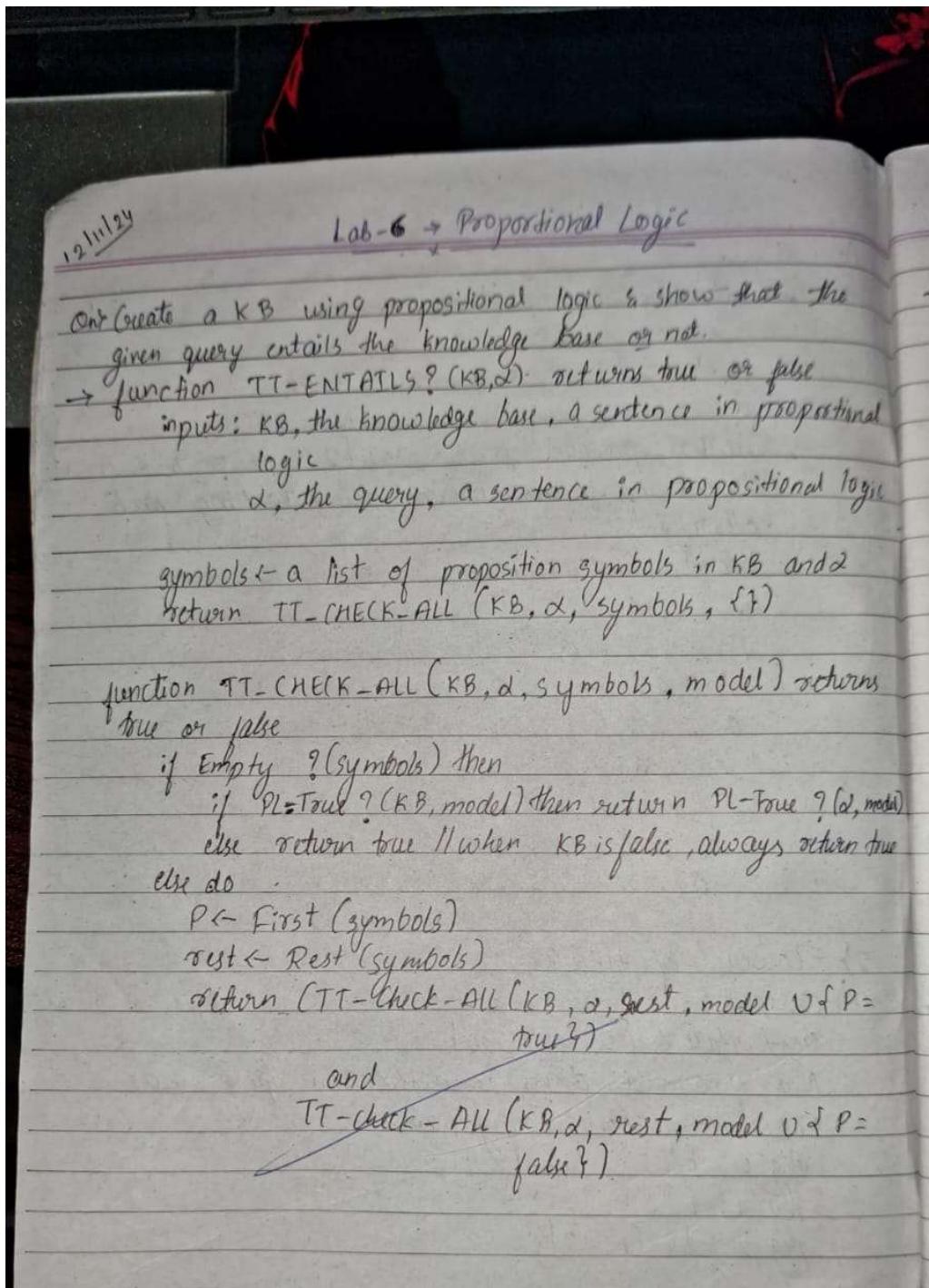
**Output :**

```
✉ The best position found is: [5 2 0 7 4 1 3 6]
The number of queens that are not attacking each other is: 7.0
```

## Program 6

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

**Algorithm:**



\* Propositional Inference: Enumeration Method.  
 ex:-  $\alpha = A \vee B$        $KB = (A \vee C) \wedge (B \vee \neg C)$

Checking that  $KB \models \alpha$

	A	B	C	$A \vee C$	$B \vee \neg C$	$KB$	$\alpha$
	false	F	F	F	T	F	F
	false	F	T	T	F	F	F
	false	T	F	F	T	F	T
	false	T	T	T	T	T	T
	true	F	F	T	T	T	T
	true	F	T	T	F	F	T
	true	T	F	T	T	T	T
	true	T	T	T	T	T	T

88Q  
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 ( $\alpha$ ): A ∨ B
table["Alpha ( $\alpha$ )"] = table["A"] | table["B"]

# Define a highlighting function
def highlight_rows(row):
    if row["KB"] and row["Alpha ( $\alpha$ )"]:
        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 ( $\alpha$ )
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:**

*LA B-7*

\* Implement unification in FOL

\* Algorithm :- Unify ( $\Psi_1, \Psi_2$ )

Step.1 : If  $\Psi_1$  or  $\Psi_2$  is a variable or constant, then  
a) If  $\Psi_1$  or  $\Psi_2$  are identical, then return NIL.  
b) Else if  $\Psi_1$  is a variable,  
    a. Then if  $\Psi_1$  occurs in  $\Psi_2$ , then return FAILURE.  
    b. Else return  $\{(\Psi_2/\Psi_1)\}$ .  
c) Else if  $\Psi_2$  is a variable,  
    a. If  $\Psi_2$  occurs in  $\Psi_1$ , then return FAILURE.  
    b. Else return  $\{(\Psi_1/\Psi_2)\}$ .  
d) Else return FAILURE.

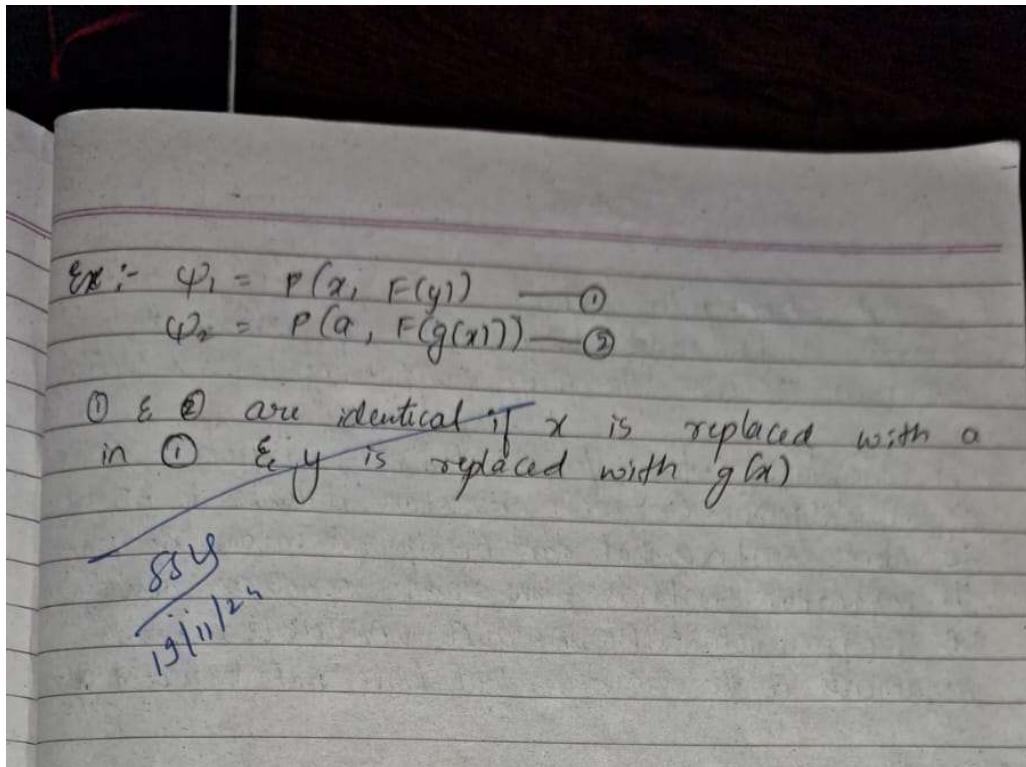
Step.2 : If the initial predicate symbol in  $\Psi_1$  &  $\Psi_2$  are not same, then return FAILURE.

Step.3 : If  $\Psi_1$  and  $\Psi_2$  have a 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 in  $\Psi_1$ ,  
    a) Call unify function with the  $i^{th}$  element of  $\Psi_1$  &  $i^{th}$  element of  $\Psi_2$ , and put the result into S.  
    b) If S = failure then returns FAILURE.  
    c) If  $S \neq NIL$  then do,  
        a. Apply S to the remainder of both 1 & 2.  
        b. SUBST = APPEND (S, SUBST).

Step 6 : Return SUBST.



**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 Chaining in FOL:

Qn) Create a knowledge base consisting of first order logic statements & prove the given query using Forward chaining

→ A conceptually straightforward, but very inefficient, forward chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences & the atomic sentences already in KB. The function STANDARDIZE-VARIABLES in its arguments with new ones that have not been used before.

A Algorithm:-

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

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

local variables : new, the new sentences inferred on each iteration.

repeat until new is empty

new  $\leftarrow \emptyset$

for each rule in KB do

$(p_1 \wedge \dots \wedge p_n \rightarrow q) \leftarrow \text{STANDARDIZE-VARIABLES}$   
(rule)

for each  $\theta$  such that  $\text{SUBST}(\theta, p_1 \wedge \dots \wedge p_n) =$   
 $\text{SUBST}(\theta, p'_1 \wedge \dots \wedge p'_n)$

for some  $p'_1, \dots, p'_n$  in KB

$q' \leftarrow \text{SUBST}(\theta, q)$

if  $q'$  does not unify with some sentence  
already in KB or new then

add  $q'$  to new

$\theta \leftarrow \text{UNIFY}(q'; \theta)$

if \$ is not fail then return \$  
add new to KB  
return false.

\* Ex 1

It is a crime for an American to sell weapons to hostile nations

Let's say p, q, r, s are variables

American (p)  $\wedge$  Weapon (q)  $\wedge$  Sells (p, q, r)  $\wedge$  Hostile (s)  $\Rightarrow$   
Criminal (p)

Country A has some missiles

$\exists x \text{ Owns}(A, x) \wedge \text{Missile}(x)$

Existential instantiation, introducing a new constant T1:

Owes (A, T1)

Missile (T1)

All of the missiles were sold to country A by Robert

$\forall x \text{ Missile}(x) \wedge \text{Owes}(A, x) \Rightarrow \text{Sells}(\text{Robert}, x, A)$

Missiles are weapons

Missile (x)  $\Rightarrow$  Weapon (x)

Enemy of America is known as hostile

$\forall x \text{ Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x)$

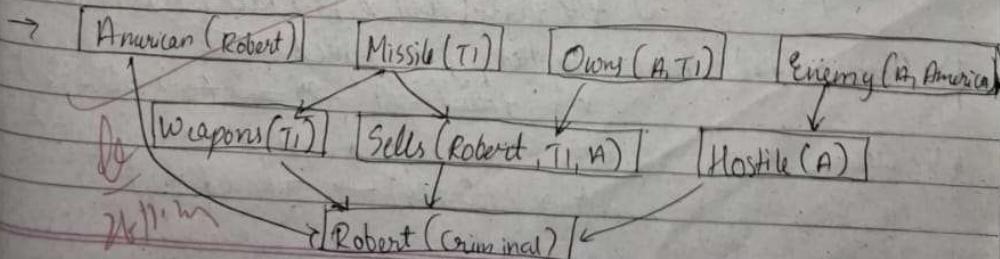
Robert is an American

American (Robert)

The country A, an enemy of America

Enemy (A, America)

To prove:  
Robert is Criminal  
Criminal (Robert)



**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:**

26/11/24

Convert FOL to Resolution.

\* Basic steps for proving a conclusion  $S$  given premises  
 Premise<sub>1</sub>, ..., Premise<sub>n</sub>  
 (all expressed in FOL)

1. Convert all sentences to CNF.
2. Negate conclusion  $S$  & 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 unifications.
  - c. If resolvent is the empty clause, a contradiction has been found (i.e.,  $S$  follows from the premises)
  - d. If not, add resolvent to the premises

If we succeed in step 4, we have proved the conclusion

Ex:- Given the KB or premises:

a) John likes all kind of food. :-  $\forall x: \text{food}(x) \rightarrow \text{likes}(\text{John}, x)$

b) Apple & vegetables are food. :-  $\text{food}(\text{Apple}) \wedge \text{food}(\text{vegetable})$

c) Anything anyone eats & not killed is food :

$$\# \forall x \forall y: \text{eats}(x, y) \wedge \neg \text{killed}(x) \rightarrow \text{food}(y)$$

d) Anil eats peanuts & still alive :-

$$\text{eats}(\text{Anil}, \text{Peanuts}) \wedge \text{alive}(\text{Anil})$$

e) Harry eats everything that Anil eats :-

$$\# \forall x: \text{eats}(\text{Anil}, x) \rightarrow \text{eats}(\text{Harry}, x)$$

f) Anyone who is alive implies not killed :-

$$\# \forall x: \text{alive}(x) \rightarrow \neg \text{killed}(x)$$

g) Anyone who is not killed implies not alive

$$\# \forall x: \neg \text{killed}(x) \rightarrow \neg \text{alive}(x)$$

Prove that resolution that  
n. John likes Peanuts

likes(John, Peanuts)

\* Proof by Resolution:

a)  $\neg \text{food}(x) \vee \text{likes}(\text{John}, x)$

b) food(Apple)

c) food(vegetables)

d)  $\neg \text{cats}(y, z) \vee \text{killed}(y) \vee \text{food}(z)$

e) eats(Anil, Peanuts)

f) alive(Anil)

g)  $\neg \text{eats}(\text{Anil}, w) \vee \text{eats}(\text{Harry}, w)$

h) killed(g)  $\vee$  alive(g)

i)  $\neg \text{alive}(k) \vee \neg \text{killed}(k)$

j) likes(John, Peanuts)

$\rightarrow \neg \text{likes}(\text{John}, \text{Peanuts})$

$\neg \text{food}(x) \vee \text{likes}(\text{John}, x)$   
{Peanuts/x}

$\neg \text{food}(\text{Peanuts})$

$\neg \text{cats}(y, z) \vee \text{killed}(y) \vee \text{food}(z)$   
{Peanuts/z}

$\neg \text{cats}(y, \text{Peanuts}) \vee \text{killed}(y)$

$\text{eats}(\text{Anil}; \text{Peanuts})$   
{Anil/y}

$\neg \text{killed}(\text{Anil})$

$\neg \text{alive}(k) \vee \neg \text{killed}(k)$   
{Anil/k}

$\neg \text{alive}(\text{Anil})$

alive(Anil)

{Hence proved}

**Code:**

```
KB = {  
    "food(Apple)": True,  
    "food(vegetables)": True,  
    "eats(Anil, Peanuts)": True,  
    "alive(Anil)": True,  
    "likes(John, X)": "food(X)", # Rule: John likes all food  
    "food(X)": "eats(Y, X) and not killed(Y)", # Rule: Anything eaten and not killed is food  
    "eats(Harry, X)": "eats(Anil, X)", # Rule: Harry eats what Anil eats  
    "alive(X)": "not killed(X)", # Rule: Alive implies not killed  
    "not killed(X)": "alive(X)", # Rule: Not killed implies alive  
}  
  
# Function to evaluate if a predicate is true based on the KB  
def resolve(predicate):  
    # If it's a direct fact in KB  
    if predicate in KB and isinstance(KB[predicate], bool):  
        return KB[predicate]  
  
    # If it's a derived rule  
    if predicate in KB:  
        rule = KB[predicate]  
        if " and " in rule: # Handle conjunction  
            sub_preds = rule.split(" and ")  
            return all(resolve(sub.strip()) for sub in sub_preds)  
        elif " or " in rule: # Handle disjunction  
            sub_preds = rule.split(" or ")  
            return any(resolve(sub.strip()) for sub in sub_preds)  
        elif "not " in rule: # Handle negation  
            sub_pred = rule[4:] # Remove "not "  
            return not resolve(sub_pred.strip())  
        else: # Handle single predicate  
            return resolve(rule.strip())  
  
    # If the predicate is a specific query (e.g., likes(John, Peanuts))  
    if "(" in predicate:  
        func, args = predicate.split("(")  
        args = args.strip(")").split(", ")  
        if func == "food" and args[0] == "Peanuts":  
            return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)")  
        if func == "likes" and args[0] == "John" and args[1] == "Peanuts":  
            return resolve("food(Peanuts)")  
  
    # Default to False if no rule or fact applies  
    return False
```

```
# Query to prove: John likes Peanuts
query = "likes(John, Peanuts)"
result = resolve(query)

# Print the result
print(f'Does John like peanuts? {"Yes" if result else "No"}')
```

**OUTPUT:**

```
⤵ Does John like peanuts? Yes
```

**Program 10**

Implement Alpha-Beta Pruning.

**Algorithm:**

3/12/24

Lab 10

## Alpha - Beta Pruning Algorithm

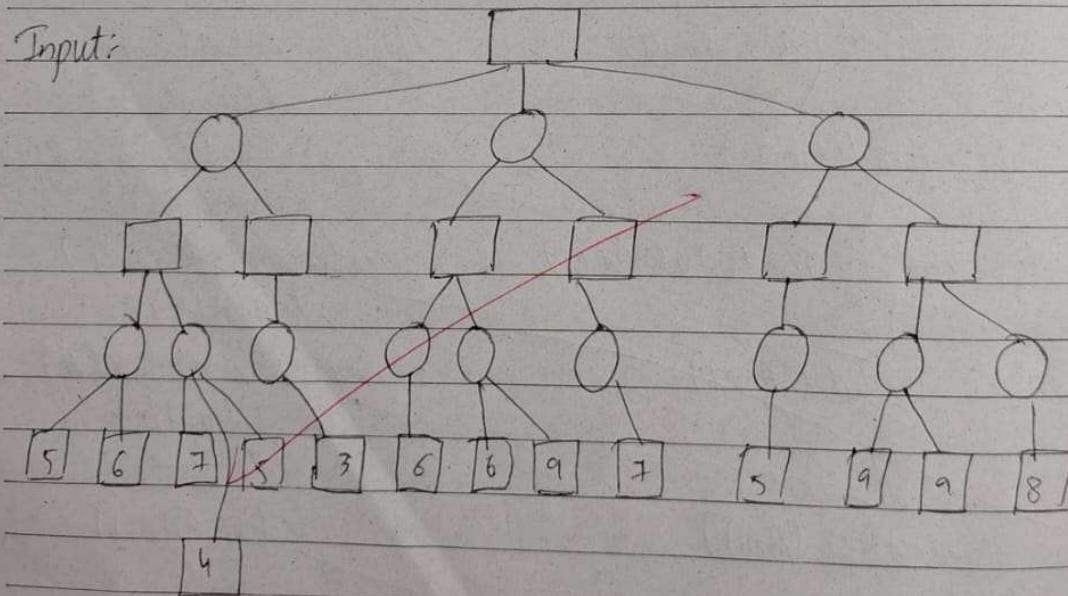
\* Algorithm:

- Alpha( $\alpha$ ) - Beta( $\beta$ ) proposes to compute find the optimal path without looking at every node in the game tree.
- Max contains Alpha( $\alpha$ ) & Min contains Beta( $\beta$ ) bound during the calculation.
- In both MIN & MAX node, we return when  $\alpha \geq \beta$  which compares with its parent node only.
- Both minimax & Alpha( $\alpha$ ) - Beta( $\beta$ ) cut-off give same path.
- Alpha( $\alpha$ ) - Beta( $\beta$ ) gives optimal solution as it takes less time to get the value for the root node.

Ex:-  max

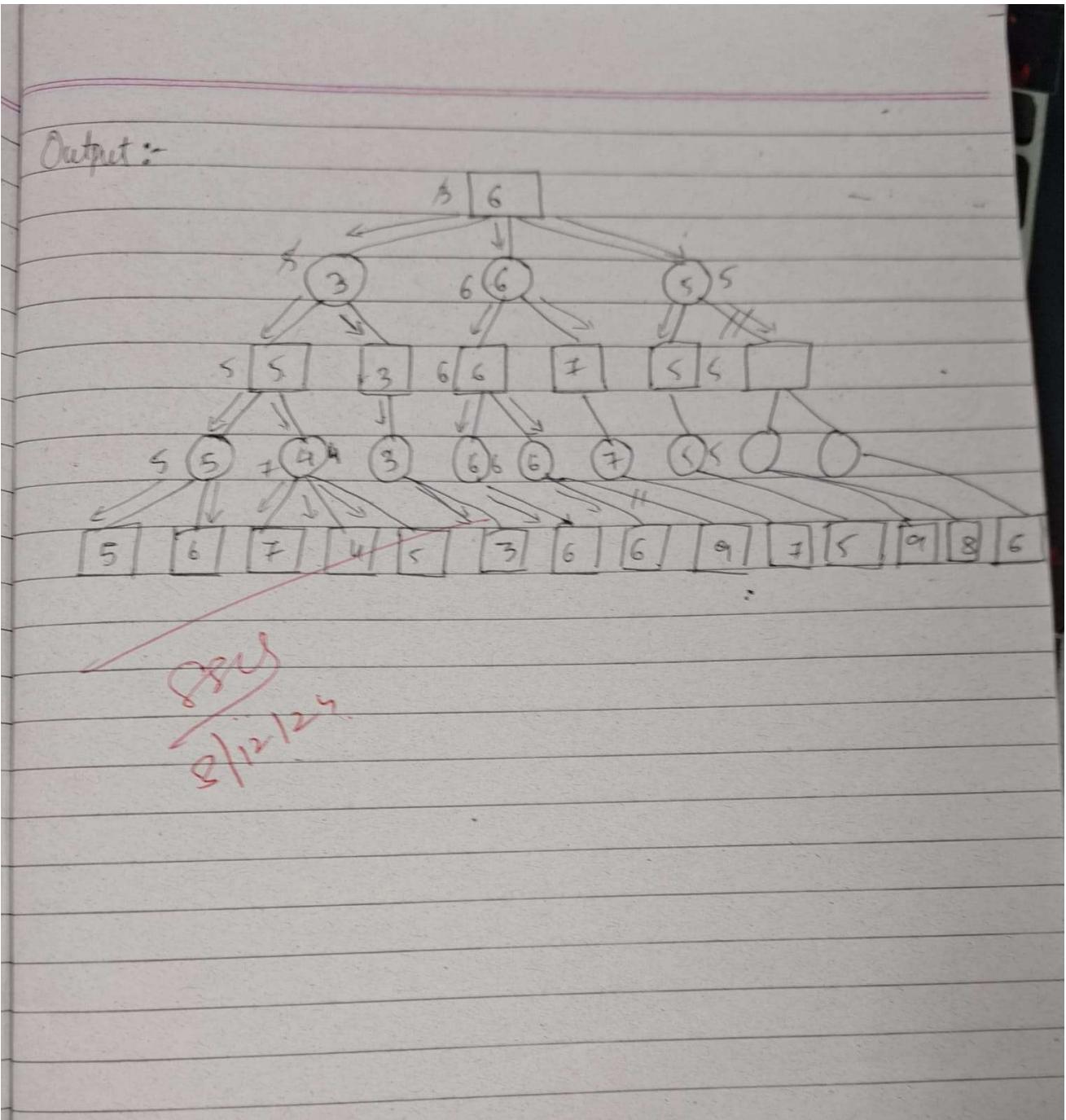
 min

Input:



$\alpha \rightarrow$  max

$\beta \rightarrow$  min



**Code:**

```
import math
def minimax(node, depth, is_maximizing):
    """
    Implement the Minimax algorithm to solve the decision tree.
    
```

Parameters:

node (dict): The current node in the decision tree, with the following structure:

```
{  
    'value': int,  
    'left': dict or None,  
    'right': dict or None  
}
```

depth (int): The current depth in the decision tree.

is\_maximizing (bool): Flag to indicate whether the current player is the maximizing player.

Returns:

int: The utility value of the current node.

"""

# Base case: Leaf node

if node['left'] is None and node['right'] is None:

```
    return node['value']
```

# Recursive case

if is\_maximizing:

```
    best_value = -math.inf  
    if node['left']:  
        best_value = max(best_value, minimax(node['left'], depth + 1, False))
```

if node['right']:  
 best\_value = max(best\_value, minimax(node['right'], depth + 1, False))

```
    return best_value
```

else:

```
    best_value = math.inf  
    if node['left']:  
        best_value = min(best_value, minimax(node['left'], depth + 1, True))
```

if node['right']:  
 best\_value = min(best\_value, minimax(node['right'], depth + 1, True))

```
    return best_value
```

# Example usage

decision\_tree = {

```
    'value': 5,  
    'left': {
```

'value': 6,

'left': {

'value': 7,

'left': {

'value': 4,

'left': None,

'right': None

},

'right': {

'value': 5,

'left': None,

```

        'right': None
    }
},
'right': {
    'value': 3,
    'left': {
        'value': 6,
        'left': None,
        'right': None
    },
    'right': {
        'value': 9,
        'left': None,
        'right': None
    }
}
},
'right': {
    'value': 8,
    'left': {
        'value': 7,
        'left': {
            'value': 6,
            'left': None,
            'right': None
        },
        'right': {
            'value': 9,
            'left': None,
            'right': None
        }
    },
    'right': {
        'value': 8,
        'left': {
            'value': 6,
            'left': None,
            'right': None
        },
        'right': None
    }
}
}
}

# Find the best move for the maximizing player
best_value = minimax(decision_tree, 0, True)
print(f"The best value for the maximizing player is: {best_value}")

```

**OUTPUT:**

```
→ The best value for the maximizing player is: 6
```