

**VISVESVARAYA TECHNOLOGICAL  
UNIVERSITY**

“JnanaSangama”, Belgaum -590014, Karnataka.



**LAB REPORT  
on**

**Artificial Intelligence (23CS5PCAIN)**

*Submitted by*

**ANKIT SINGH BHATTI (1BM22CS353)**

*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 **ANKIT SINGH BHATTI (1BM22CS353)**, 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.

Prof. Sunayana S 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	1-12
2	7-10-2024	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	13-16
3	14-10-2024	Implement A* search algorithm	17-24
4	21-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	25-29
5	28-10-2024	Simulated Annealing to Solve 8-Queens problem	30-32
6	11-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	33-36
7	2-12-2024	Implement unification in first order logic	37-41
8	2-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	42-44
9	16-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	45-48
10	16-12-2024	Implement Alpha-Beta Pruning.	49-52

Github Link: [GITHUB](#)

## Program 1

Implement Tic - Tac - Toe Game  
Implement vacuum cleaner agent

Algorithm:

URBAN 24, 9, 24  
EDGE

**LAB 1 : Explore Python : Programming language and Implement the game of TIC TAC TOE.**

**Algorithm :** Generate (choice)

**Input :** choice = 'X' or 'O' to be generated on the board

**Output :** NULL (sets the array value)

```

for i in range(9):
    a = random.randint(0, 2)
    b = random.randint(0, 2)
    if (array[a][b] != "X" and
        array[a][b] != "O"):
        array[a][b] = choice
    if (a == 0 and b == 0):
        return None
print("No empty positions")

```

**Algorithm :** Player (choice)

**Input :** choice = 'X' or 'O' to be given to the player

**Output :** NULL (sets the array value)

```

row = int(input("Enter Row value (0-2):"))
col = int(input("Enter Column value (0-2):"))

if 0 <= row < 3 and 0 <= column < 3:
    if (array[row][column] == 'X' or
        array[row][column] == 'O'):
        print("Invalid row or column")
    else:
        array[row][column] = choice

```

Algorithm : checkwinner()

Input : None

Output : Winner of the game  
i.e 'x' or 'o'

for line(0 to n) array:

if all 'x' or 'o' == board[i]:

if line[0] == line[1] == line[2]

(center pos and line[0] != 0):

return line[0]

: (0) same as 0?

for col in range(3):

(0,1,2):

if array[0][col] != array[1][col]

: 'o' == array[2][col] and array[0][col]

and (array[0][col] == 0):

return array[0][col]

(empty after all loops)

return 0

: (0,0) position : empty spot

Algorithm : Pctactoe():

Input : None, with col

Output : (winner) of the game

choice = random.choice(['x', 'o'])

print("You are", choice)

if choice == 'x':

choice = generate('o')

else: - Computer's turn if

choice = generate('x')

print(array)

for i in range(1,10):

if choice == "x":

number botomotu generated ("0") : S.001

else if (A == 3) then

generate ("x")

player (choice)

return A or print (array)

else if (A == 4) then

if (winner == checkWinner (C))

PF: winner == true then

else print ("The winner is: " + winner)

break; print

A. print ("good work")

else if (A == 5) then

(a) print ("It's a draw!")

B. print ("bad")

24/9/2024

at 10:00 AM

at 10:00 AM

#### OUTPUT:

You're optional form ]9 C-

[C 0] X.10 29 X10001 17

[9 X1000 next.]

[End of 0 list.]

The winner is 0.

and you have won 2000 words 2023/09/18

and you have won 2000 words 2023/09/18

LAB 2 : Algorithm for Automated Vacuum  
cleaner. (2 & 4 QUADRANTS)

( " " ) 330 1430

Algorithm: 2 modes:

( 09.00 ) 19/01/24

1) Input: Read room\_A &amp; room\_B states

2) Initialize :

→ Set current\_location to 'A' .

→ set total\_cost to 0 .

→ Define goal\_state as both rooms  
being clean .

3) Define clean\_room function :

→ If current location ps 'A':

→ If room\_A dirty(0) :

Clean room\_A

Add 2 to cost

move to room\_B

→ If current\_location ps 'B'

→ If room\_B dirty(0) :

Clean room\_B

Add 2 to cost

4) Define check\_goal\_state function :

→ If both rooms are clean , return  
goal achieved .

→ otherwise , call clean\_room .

5) Run loop :

→ continue checking and cleaning

6) Output: Display total cost and final  
state .

Output: 1. Initial room status [A,B] : (1,1)

enter state for 'A' is 1.

enter state for 'B' is 0

Vacuum is placed in location A

location A is dirty

Cleaning location A cost is 2

cost for suck is 2 units

Moving right to location B

Vacuum is placed in location B

location B is already clean (0)

Both rooms are clean!

Goal state reached: [A, 0, B, 0]

Total cost : 2 units

Algorithm: 4 Quadrants (both rooms)

A = 0, B = 0

1) Input : Read 1 room-A & room-B states.

A & B room-C & room-D states

2) Initialize :

→ Set current location to either 'A', 'B', 'C' or 'D'.

→ Set total cost to 0.

→ Define goal state as both room being clean.

If input start room as A.

3) Define clean room function :

→ If current location is A:

i) If room A is dirty:

→ Make it clean room-A

→ Add 2 to cost

→ move to A+1 or vizi

(i.e., if A=0, B=1 then move to A+1)

→ clean room-B

4) Define check\_goal\_status function:

→ If all rooms are clean, return 0 goal achieved.

If not → otherwise, i.e., clean room

5) Run loop: An additional input + 1

→ continuous checking for change in

→ stated at each iteration

→ iteration after interval of time

6) Output: Display total cost and final

state of the house.

Output: A state : D, B state : C, C state : B, D state : A

Enter A state : 1

Enter B state : 0, C state : 0, D state : 0

Enter C state : 0

Enter D state : 1 target : target (1)

Vacuum qs placed in A

Location A qs dirty : 0 cost : 0

Cost for 'suck' in A : 0

Moving to B

Vacuum qs placed in B

B qs clean : 1 cost : 0

Moving to C

Vacuum qs placed in C

C qs clean : 1 cost : 0

Moving to D

Vacuum qs placed in D

Location D qs dirty

Cost for 'suck' : 2 units

All rooms are clean, C

Goal state : [A, 0, B, 0, C, 0, D, 0]

Total cost : 4

S8  
3/10/2024

```

Code: TIC TAC TOE
import numpy as np
import random

array = np.full((3, 3), '-', dtype=object)

def generate(choice):
    for _ in range(9):
        a = random.randint(0, 2)
        b = random.randint(0, 2)
        if array[a][b] != "X" and array[a][b] != "O":
            array[a][b] = choice
            return
    print("No empty positions left! It's a draw")

def player(choice):
    row = int(input("Enter the Row Value (0-2): "))
    column = int(input("Enter the Column Value (0-2): "))
    if 0 <= row < 3 and 0 <= column < 3:
        if array[row][column] == "X" or array[row][column] == "O":
            print("Invalid row or column index. Position already taken.")
        else:
            array[row][column] = choice
    else:
        print("Invalid row or column index.")

def check_winner():
    for line in array:
        if line[0] == line[1] == line[2] and line[0] != '-':
            return line[0]
    for col in range(3):
        if array[0][col] == array[1][col] == array[2][col] and array[0][col] != '-':
            return array[0][col]
    if array[0][0] == array[1][1] == array[2][2] and array[0][0] != '-':
        return array[0][0]
    if array[0][2] == array[1][1] == array[2][0] and array[0][2] != '-':
        return array[0][2]
    return 0

choice = random.choice(["X", "O"])
print("You are", choice)

if choice == "X":
    generate("O")
else:
    generate("X")

```

```

print(array)

for i in range(1, 10):
    player(choice)
    if choice == "X":
        generate("O")
    else:
        generate("X")
    print(array)
    winner = check_winner()
    if winner != 0:
        print(f"The winner is: {winner}")
        break

```

OUTPUT:

```

You are X
[['O']]
[[-]]
[[-]]
Enter the Row Value (0-2): 0
Enter the Column Value (0-2): 0
[['X' 'O' '-']]
[[-]]
[[-]]
[['X' 'O' '-']]
[[-]]
[[-]]
Enter the Row Value (0-2): 1
Enter the Column Value (0-2): 1
[['X' 'O' '-']]
[[-'X' '-']]
[[-]]
[['X' 'O' '-']]
[[-'X' '-']]
[[-'O' '-']]
Enter the Row Value (0-2): 2
Enter the Column Value (0-2): 2
[['X' 'O' '-']]
[[-'X' '-']]
[[-'-' 'X']]
The winner is: X

```

```

Code: VACCUUM WORLD (2 QUADS)
#2QUADS
room_A = int(input("Enter the state of Room A (1 for Dirty, 0 for Clean): "))
room_B = int(input("Enter the state of Room B (1 for Dirty, 0 for Clean): "))
current_location = 'A'
goal_state = ['A', 0, 'B', 0]
total_cost = 0

# Step 4: Define a function to clean the room
def clean_room():
    global room_A, room_B, current_location, total_cost
    # Check the current location
    if current_location == 'A':
        print("Vacuum is placed in Location A")
        if room_A == 1:
            print("Location A is Dirty")
            print("Cleaning Location A...")
            room_A = 0 # Clean the room
            total_cost += 2
            print("Location A has been Cleaned")
            print("COST for SUCK in Location A: 2 units")
    else:
        print("Location A is already clean")
    # Move to Location B (no cost for moving)
    print("Moving right to Location B")
    current_location = 'B'
    if current_location == 'B':
        print("Vacuum is placed in Location B")
        if room_B == 1:
            print("Location B is Dirty")
            print("Cleaning Location B...")
            room_B = 0 # Clean the room
            total_cost += 2 # Cost of cleaning (SUCK action)
            print("Location B has been Cleaned")
            print("COST for SUCK in Location B: 2 units")
    else:
        print("Location B is already clean")

# Step 5: Keep checking for the goal state
def check_goal_state():
    if room_A == goal_state[1] and room_B == goal_state[3]:
        print("Both rooms are clean! Goal state reached:", goal_state)
        return True # Goal state reached, stop the vacuum cleaner
    else:
        clean_room() # Continue cleaning until the goal state is reached
        return False

```

```

# Step 6: Run the vacuum cleaner
goal_reached = False # A flag to track if the goal is reached
while not goal_reached: # Loop until the goal state is reached
    goal_reached = check_goal_state()

# Final output after cleaning
print("Final goal state:", goal_state)
print("Total cost incurred:", total_cost, "units")

```

OUTPUT:

```

Enter the state of Room A (1 for Dirty, 0 for Clean): 1
Enter the state of Room B (1 for Dirty, 0 for Clean): 1
Vacuum is placed in Location A
Location A is Dirty
Cleaning Location A...
Location A has been Cleaned
COST for SUCK in Location A: 2 units
Moving right to Location B
Vacuum is placed in Location B
Location B is Dirty
Cleaning Location B...
Location B has been Cleaned
COST for SUCK in Location B: 2 units
Both rooms are clean! Goal state reached: ['A', 0, 'B', 0]
Final goal state: ['A', 0, 'B', 0]
Total cost incurred: 4 units

```

Code: VACCUUM WORLD(4 QUADS)

```

# 4-Quads Vacuum Cleaner Problem
room_A = int(input("Enter the state for room A (1 for Dirty, 0 for Clean): "))
room_B = int(input("Enter the state for room B (1 for Dirty, 0 for Clean): "))
room_C = int(input("Enter the state for room C (1 for Dirty, 0 for Clean): "))
room_D = int(input("Enter the state for room D (1 for Dirty, 0 for Clean): "))

current_location = 'A'
goal_state = ['A', 0, 'B', 0, 'C', 0, 'D', 0]
total_cost = 0

def clean_room():
    global room_A, room_B, room_C, room_D, current_location, total_cost
    if current_location == 'A':
        print("Vacuum is placed in Location A")
        if room_A == 1:
            print("Location A is Dirty")
            print("Cleaning Location A...")
            room_A = 0
    
```

```

total_cost += 2
print("Location A has been Cleaned")
print("COST for SUCK in Location A: 2 units")
else:
    print("Location A is already clean")
    print("Moving right to Location B")
    current_location = 'B'
elif current_location == 'B':
    print("Vacuum is placed in Location B")
    if room_B == 1:
        print("Location B is Dirty")
        print("Cleaning Location B...")
        room_B = 0
        total_cost += 2
        print("Location B has been Cleaned")
        print("COST for SUCK in Location B: 2 units")
    else:
        print("Location B is already clean")
        print("Moving down to Location C")
        current_location = 'C'
elif current_location == 'C':
    print("Vacuum is placed in Location C")
    if room_C == 1:
        print("Location C is Dirty")
        print("Cleaning Location C...")
        room_C = 0
        total_cost += 2
        print("Location C has been Cleaned")
        print("COST for SUCK in Location C: 2 units")
    else:
        print("Location C is already clean")
        print("Moving right to Location D")
        current_location = 'D'
elif current_location == 'D':
    print("Vacuum is placed in Location D")
    if room_D == 1:
        print("Location D is Dirty")
        print("Cleaning Location D...")
        room_D = 0
        total_cost += 2
        print("Location D has been Cleaned")
        print("COST for SUCK in Location D: 2 units")
    else:
        print("Location D is already clean")

def check_goal_state():
    if room_A == goal_state[1] and room_B == goal_state[3] and room_C == goal_state[5] and

```

```

room_D == goal_state[7]:
    print("All rooms are clean! Goal state reached:", goal_state)
    return True
else:
    clean_room()
    return False

goal_reached = False
while not goal_reached:
    goal_reached = check_goal_state()

print("Final goal state:", goal_state)
print("Total cost incurred:", total_cost, "units")

```

**OUTPUT:**

```

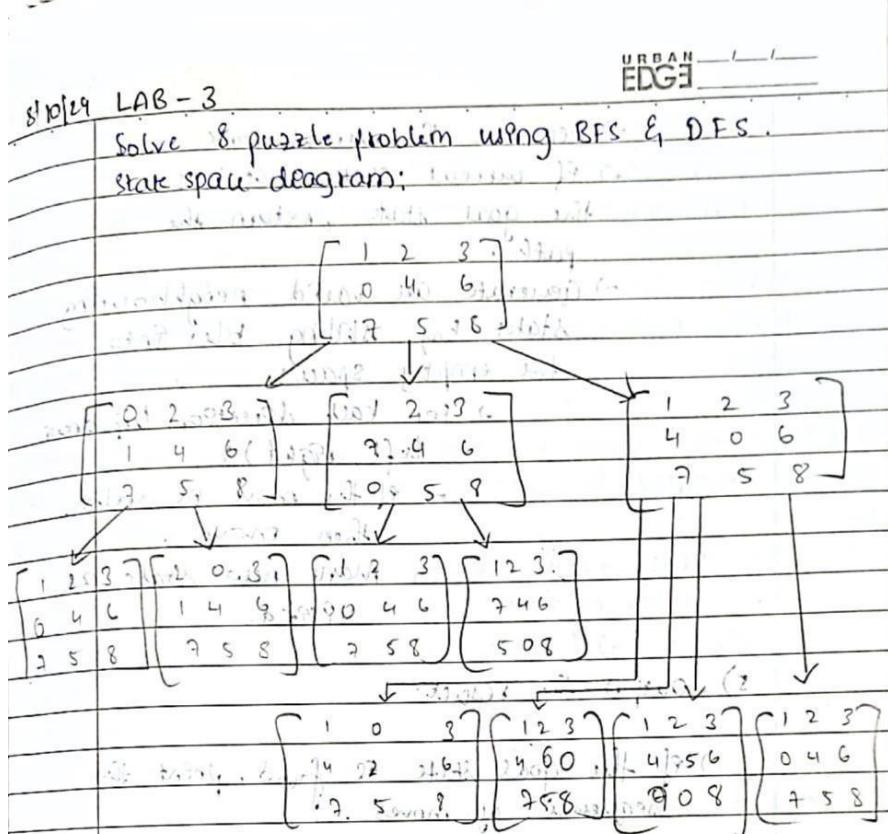
Enter the state for room A (1 for Dirty, 0 for Clean): 1
Enter the state for room B (1 for Dirty, 0 for Clean): 0
Enter the state for room C (1 for Dirty, 0 for Clean): 1
Enter the state for room D (1 for Dirty, 0 for Clean): 1
Vacuum is placed in Location A
Location A is Dirty
Cleaning Location A...
Location A has been Cleaned
COST for SUCK in Location A: 2 units
Moving right to Location B
Vacuum is placed in Location B
Location B is already clean
Moving down to Location C
Vacuum is placed in Location C
Location C is Dirty
Cleaning Location C...
Location C has been Cleaned
COST for SUCK in Location C: 2 units
Moving right to Location D
Vacuum is placed in Location D
Location D is Dirty
Cleaning Location D...
Location D has been Cleaned
COST for SUCK in Location D: 2 units
All rooms are clean! Goal state reached: ['A', 0, 'B', 0, 'C', 0, 'D', 0]
Final goal state: ['A', 0, 'B', 0, 'C', 0, 'D', 0]
Total cost incurred: 6 units

```

## **Program 2**

## SOLVE 8 PUZZLE PROBLEM USING BFS AND DFS

### Algorithm:



8. puzzle algorithm: using the IP

~~stomach, bladder, heart, brain~~

1) Define Goal State is (1, 2, 3, 4, 5, 6, 7, 8, 0)

2) Input Starting String:

→ Prompt user for the optimal configuration

of the puzzle as 9 numbers (0-8).

### 3) Implemente BFS:

→ create a queue and add the starting state with an empty path

$\Rightarrow$  Create a set to track visited states.

4) BFS:

→ while the queue is not empty:

- Dequeue the current state
- If current state matches with the goal state, return the path.
- Generate all valid neighbouring states by sliding tiles onto the empty space:
  - For each direction (up, down, left, right):
  - If the move is valid, then move.
  - Mark new state as visited.

### 5) Output the result:

→ If the goal state is found, print the sequence of moves.

→ If the given user is empty / and the goal is not reached, indicate

~~(10) Solution not found~~

~~Stop 2024~~

: print position? right (2)

~~position 2024 is the target~~

~~(2) of solution is 2024. Hence all g~~

~~(2) 2024 is the target~~

~~right is the final output of stored c~~

~~on, taking the user 2024~~

~~2024 is the final output of stored c~~

~~(2) 2024~~

~~(2) 2024 is the final output of stored c~~

Code:

```
from collections import deque
```

```
GOAL_STATE = (1, 2, 3, 4, 5, 6, 7, 8, 0)
```

```
def find_empty(state):  
    return state.index(0)
```

```
def get_neighbors(state):  
    neighbors = []  
    empty_index = find_empty(state)  
    row, col = divmod(empty_index, 3)  
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  
    for dr, dc in directions:  
        new_row, new_col = row + dr, col + dc  
        if 0 <= new_row < 3 and 0 <= new_col < 3:  
            new_index = new_row * 3 + new_col  
            new_state = list(state)  
            new_state[empty_index], new_state[new_index] = new_state[new_index],  
            new_state[empty_index]  
            neighbors.append(tuple(new_state))  
    return neighbors
```

```
def bfs(initial_state):  
    queue = deque([(initial_state, [])])  
    visited = set()  
    visited.add(initial_state)  
    visited_count = 1 # Initialize visited count  
    while queue:  
        current_state, path = queue.popleft()  
        if current_state == GOAL_STATE:  
            return path, visited_count # Return path and count  
        for neighbor in get_neighbors(current_state):  
            if neighbor not in visited:  
                visited.add(neighbor)  
                queue.append((neighbor, path + [neighbor]))  
                visited_count += 1 # Increment visited count  
    return None, visited_count # Return count if no solution found
```

```
def input_start_state():  
    print("Enter the starting state as 9 numbers (0 for the empty space):")  
    input_state = input("Format: 1 2 3 4 5 6 7 8 0\n")  
    numbers = list(map(int, input_state.split()))  
    if len(numbers) != 9 or set(numbers) != set(range(9)):  
        print("Invalid input. Please enter numbers from 0 to 8 with no duplicates.")  
    return input_start_state()
```

```

return tuple(numbers)

def print_matrix(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])

if __name__ == "__main__":
    initial_state = input_start_state()
    print("Initial state:")
    print_matrix(initial_state)
    solution, visited_count = bfs(initial_state)
    print(f"Number of states visited: {visited_count}")
    if solution:
        print("\nSolution found with the following steps:")
        for step in solution:
            print_matrix(step)
            print()
    else:
        print("No solution found.")

```

#### OUTPUT:

Enter the starting state as 9 numbers (0 for the empty space):  
Format: 1 2 3 4 5 6 7 8 0  
Initial state:  
[1, 2, 3]  
[4, 5, 6]  
[7, 0, 8]  
Number of states visited: 3

Solution found with the following steps:  
[1, 2, 3]  
[4, 5, 6]  
[7, 8, 0]

### Program 3

Implement A\* SEARCH TO SOLVE 8 PUZZLE PROBLEM (MISSING TILES AND MANHATTAN DISTANCE APPROACH)

Algorithm:

URGENT  
EDGE

Algorithm : A\* search

- 1) Place the start node  $p_0$  in the open list.
- 2) Check if the open list  $P_s$  is empty or not.  
If  $P_s$  is empty then return failure and stop.
- 3) Select the node from the open list which has the smallest value of evaluation function  $f(n) = g(n) + h(n)$ .
- 4) Compute the function  $f(n) = g(n) + h(n)$   
where  $g(n)$  is the function value and  $h(n)$  is the level (and)  $h(n)$  is the heuristic value.
- 5) For each node, expand node  $n$  to all of its successors and calculate  $f(n)$  for each corresponding successor.
- 6) The node with the smallest value is then considered for expansion and step 5 is repeated until the goal node is obtained.
- 7) Exit after goal state is obtained.

**OUTPUT : MISSING TILES**

Enter the start state: 2 8 3 1 6 4 7 0 5  
number of states visited: 12  
Solution found at depth: 5 with cost: 5

Soln steps:

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 0 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 2 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 3 & 7 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

OUTPUT: (Manhattan Approach)

Enter the start state : 2 8 3 1 6 4 7 0 5

Number of states visited : 8400

Solution found at depth : 5 (with cost: 5)  
soln steps:

$$\begin{bmatrix} 2 & 0 & 8 & 3 \\ 1 & 6 & 4 & 7 \\ 5 & 7 & 6 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 0 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 2 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 3 & 3 & 0 & 7 \\ 0 & 8 & 4 & 6 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 3 & 3 & 0 & 7 \\ 7 & 8 & 0 & 4 & 5 \end{bmatrix}$$

15/10/2024

(2331 + 2332) = 46640

20 6 0 1 8 3 8 ; 30000 20000 10000

Q1 ; 60000 30000 is solution

Q2 ; 60000 2 ; depth to blank intial

3

10000 100?

## Code: MISSING TILES APPROACH

```
import heapq

# Define the goal state as a tuple
GOAL_STATE = (1, 2, 3, 8, 0, 4, 7, 6, 5)

# Function to find the index of the empty space (0)
def find_empty(state):
    return state.index(0)

# Heuristic function that counts the number of mismatched tiles
def mismatched_tiles(state):
    return sum(1 for i, tile in enumerate(state) if tile != 0 and tile != GOAL_STATE[i])

# Function to get neighbors (all possible moves from current state)
def get_neighbors(state):
    neighbors = []
    empty_index = find_empty(state)
    row, col = divmod(empty_index, 3)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # up, down, left, right
    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[empty_index], new_state[new_index] = new_state[new_index], new_state[empty_index]
            neighbors.append(tuple(new_state))
    return neighbors

# A* algorithm implementation using mismatched tiles heuristic
def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [])) # (priority, state, path)
    visited = set()
    visited.add(initial_state)
    visited_count = 1 # Initialize visited count
    cost_so_far = {initial_state: 0} # Tracks the cost to reach each state

    while priority_queue:
        priority, current_state, path = heapq.heappop(priority_queue)

        if current_state == GOAL_STATE:
            depth = len(path)
            return path, visited_count, depth, cost_so_far[current_state]

    # Explore neighbors
```

```

for neighbor in get_neighbors(current_state):
    new_cost = cost_so_far[current_state] + 1 # Cost to move to the neighbor

    if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
        cost_so_far[neighbor] = new_cost
        priority = new_cost + mismatched_tiles(neighbor) # A* f = g + h
        heapq.heappush(priority_queue, (priority, neighbor, path + [neighbor]))
    if neighbor not in visited:
        visited.add(neighbor)
        visited_count += 1 # Increment visited count

return None, visited_count, 0, 0 # No solution found

# Function to take user input for the initial state
def input_start_state():
    print("Enter the starting state as 9 numbers (0 for the empty space):")
    input_state = input("Format: 1 2 3 4 5 6 7 8 0\n")
    numbers = list(map(int, input_state.split()))
    if len(numbers) != 9 or set(numbers) != set(range(9)):
        print("Invalid input. Please enter numbers from 0 to 8 with no duplicates.")
        return input_start_state()
    return tuple(numbers)

# Function to print the state as a 3x3 matrix
def print_matrix(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])

# Main function
if __name__ == "__main__":
    initial_state = input_start_state()
    print("Initial state:")
    print_matrix(initial_state)

    # Run A* algorithm
    solution, visited_count, depth, cost = a_star(initial_state)

    print(f"Number of states visited: {visited_count}")
    if solution:
        print(f"Solution found at depth: {depth} with cost: {cost}")
        print("\nSolution steps:")
        for step in solution:
            print_matrix(step)
            print()
    else:
        print("No solution found.")

OUTPUT:

```

Enter the starting state as 9 numbers (0 for the empty space):

Format: 1 2 3 4 5 6 7 8 0

2 8 3 1 6 4 7 0 5

Initial state:

(2, 8, 3)

(1, 6, 4)

(7, 0, 5)

Number of states visited: 12

Solution found at depth: 5 with cost: 5

Solution steps:

(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

```
import heapq
```

```
# Define the goal state as a tuple
```

```
GOAL_STATE = (1, 2, 3, 8, 0, 4, 7, 6, 5)
```

```
# Function to find the index of the empty space (0)
```

```
def find_empty(state):
```

```
    return state.index(0)
```

```
# Heuristic function that calculates the Manhattan distance
```

```
def manhattan_distance(state):
```

```
    distance = 0
```

```
    for i, tile in enumerate(state):
```

```
        if tile != 0: # Skip the blank tile (0)
```

```
            # Calculate the correct position (row, col) in the goal state
```

```

correct_row, correct_col = divmod(tile - 1, 3)
# Calculate the current position (row, col)
current_row, current_col = divmod(i, 3)
# Add the Manhattan distance for this tile
distance += abs(correct_row - current_row) + abs(correct_col - current_col)
return distance

# Function to get neighbors (all possible moves from current state)
def get_neighbors(state):
    neighbors = []
    empty_index = find_empty(state)
    row, col = divmod(empty_index, 3)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # up, down, left, right
    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[empty_index], new_state[new_index] = new_state[new_index],
            new_state[empty_index]
            neighbors.append(tuple(new_state))
    return neighbors

# A* algorithm implementation using Manhattan distance heuristic
def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [])) # (priority, state, path)
    visited = set()
    visited.add(initial_state)
    visited_count = 1 # Initialize visited count
    cost_so_far = {initial_state: 0} # Tracks the cost to reach each state

    while priority_queue:
        priority, current_state, path = heapq.heappop(priority_queue)

        if current_state == GOAL_STATE:
            depth = len(path)
            return path, visited_count, depth, cost_so_far[current_state]

        # Explore neighbors
        for neighbor in get_neighbors(current_state):
            new_cost = cost_so_far[current_state] + 1 # Cost to move to the neighbor

            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost + manhattan_distance(neighbor) # A* f = g + h
                heapq.heappush(priority_queue, (priority, neighbor, path + [neighbor]))

```

```

        if neighbor not in visited:
            visited.add(neighbor)
            visited_count += 1 # Increment visited count

    return None, visited_count, 0, 0 # No solution found

# Function to take user input for the initial state
def input_start_state():
    print("Enter the starting state as 9 numbers (0 for the empty space):")
    input_state = input("Format: 1 2 3 4 5 6 7 8 0\n")
    numbers = list(map(int, input_state.split()))
    if len(numbers) != 9 or set(numbers) != set(range(9)):
        print("Invalid input. Please enter numbers from 0 to 8 with no duplicates.")
        return input_start_state()
    return tuple(numbers)

# Function to print the state as a 3x3 matrix
def print_matrix(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])

# Main function
if __name__ == "__main__":
    initial_state = input_start_state()
    print("Initial state:")
    print_matrix(initial_state)

    # Run A* algorithm
    solution, visited_count, depth, cost = a_star(initial_state)

    print(f"Number of states visited: {visited_count}")
    if solution:
        print(f"Solution found at depth: {depth} with cost: {cost}")
        print("\nSolution steps:")
        for step in solution:
            print_matrix(step)
            print()
    else:
        print("No solution found.")

```

#### OUTPUT:

```

Enter the starting state as 9 numbers (0 for the empty space):
Format: 1 2 3 4 5 6 7 8 0
2 8 3 1 6 4 7 0 5
Initial state:
(2, 8, 3)
(1, 6, 4)

```

(7, 0, 5)

Number of states visited: 34

Solution found at depth: 5 with cost: 5

Solution steps:

(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)

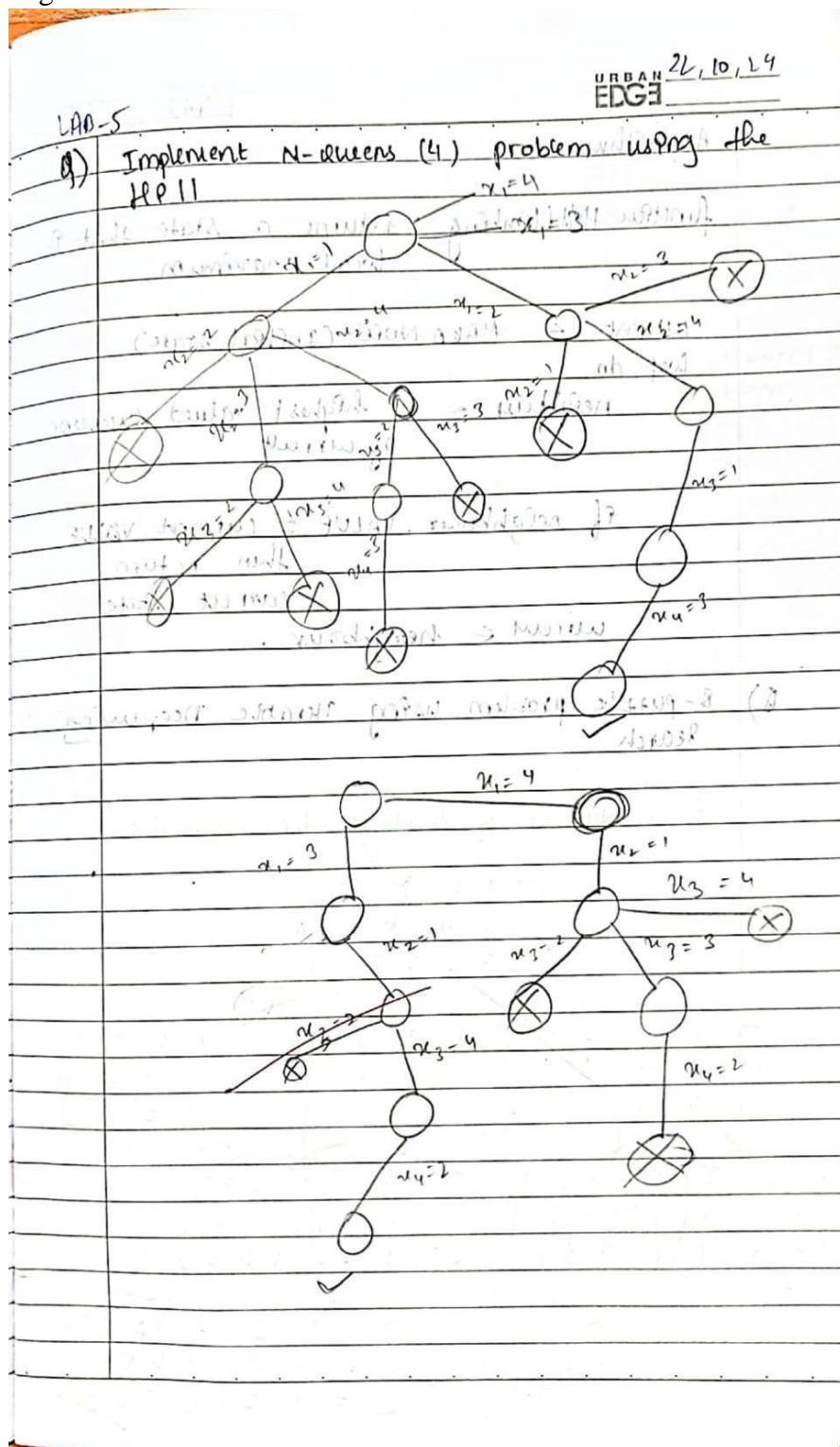
(8, 0, 4)

(7, 6, 5)

#### Program 4

Implement N-QUEENS PROBLEM USING HILL CLIMBING ALGORITHM

Algorithm:



Algorithm:

function Hill Climbing : returns a state that is local maximum

current  $\leftarrow$  MAKE-NODE (initial state)

loop do

neighbour  $\leftarrow$  a highest valued successor  
of current

if neighbour.VALUE  $\leq$  current.VALUE

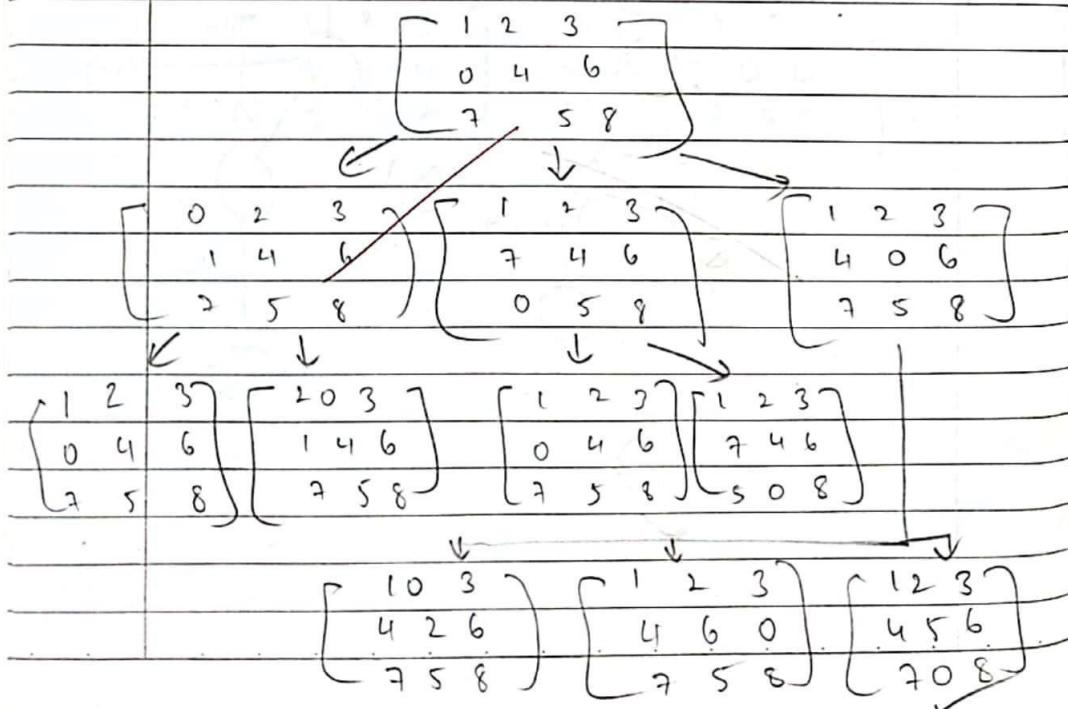
then return

current.state

current  $\leftarrow$  neighbour.

Q) 8-puzzle problem using Iterative Deepening Search

Enter Number of levels to be explored : 3



Algorithms with problem branching and backtracking

function ITDS(problem) returns a solution or failure.

for depth = 0 to max-depth do begin (1)  
    if status(result)  $\leftarrow$  DEPTH-LIMITED-SEARCH  
        then exit with result for current problem,  
    return(result); otherwise

        expanding(result) with expand(result) (2)  
        returning current problem to stack  
        + current node in stack

        problem branch stored (3)  
        problem remains to solve with new  
        + returning to start with step (2)

    else backtracking(result) (4) return  
        status(result) = failure

end loop; expansion of stack  
        + current node in stack

(current problem removed from stack  $\rightarrow$  expanded  
        + current node in stack)

(3) continuing loop

        current problem with  $O = P \cup B$

        branch problem with  $O \rightarrow P \cup B$

        expand O branching

        expand + current problem  $\rightarrow$  ITDS

        from expanded make  $O \subseteq P \cup B$

        backtrack, new node, new  $O \rightarrow$  expanding step  
        repeat

Code:

```
import random

# Function to calculate conflicts in the current board
def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

# Hill climbing algorithm
def hill_climbing(n):
    cost = 0
    while True:
        # Initialize a random board
        current_board = list(range(n))
        random.shuffle(current_board)
        current_conflicts = calculate_conflicts(current_board)

        while True:
            # Generate neighbors by moving each queen to a different position
            found_better = False

            for i in range(n):
                for j in range(n):
                    if j != current_board[i]: # Only consider different positions
                        neighbor_board = list(current_board)
                        neighbor_board[i] = j
                        neighbor_conflicts = calculate_conflicts(neighbor_board)

                        if neighbor_conflicts < current_conflicts:
                            print_board(current_board)
                            print("Current Conflicts:", current_conflicts)
                            print_board(neighbor_board)
                            print("Neighbor Conflicts:", neighbor_conflicts)

                            current_board = neighbor_board
                            current_conflicts = neighbor_conflicts
                            cost += 1
                            found_better = True
                            break
                if found_better:
                    break
            if found_better:
                break
```

```

# If no better neighbor is found, stop searching
if not found_better:
    break

# If a solution is found (zero conflicts), return the board
if current_conflicts == 0:
    return current_board, current_conflicts, cost

# Function to print the board
def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['.] * n
        row[board[i]] = 'Q' # Place a queen
        print(''.join(row))
    print("\n=====")

# Example Usage
if __name__ == "__main__":
    n = 4 # Change N here to test for different sizes
    solution, conflicts, cost = hill_climbing(n)
    print("Final Board Configuration:")
    print_board(solution)
    print("Number of Conflicts:", conflicts)
    print("Number of Steps (Cost):", cost)

```

OUTPUT:

```

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

```

Current Conflicts: 3

```

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

```

Neighbor Conflicts: 1

Final Board Configuration:

```

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

```

Number of Conflicts: 0

Number of Steps (Cost): 3

## Program 5

Implement SIMULATED ANNEALING  
Algorithm:

LAB - B

URBAN  
EDGE

simulated Annealing for the N-Queens problem.

Algorithm:

1) Define fitness function:

(create a function to calculate the number of queens that are not attacking each other)

2) Initialize the optimization problem:

→ Set up a discrete optimization problem for the N-queens

3) Execute Simulated Annealing:

→ Run the simulated Annealing algorithm to find the best configuration.

function SA(n-Queens (problem, schedule))  
returns a solution state

Inputs: problem, a problem  
schedule, a mapping from time  
to "temp".

current ← MAKE-NODE (problem, INITIAL-STATE)

for t=1 to n do

T ← schedule (t)

if T = 0 then return current

next ← a randomly selected successor of current

ΔE ← next.VALUE - current.VALUE

If ΔE > 0 then current ← next

else current ← next only with probability  $e^{\Delta E/T}$

OUTPUT: ~~Queens~~ Found using Branch and Bound

The best position found: Ps = [5 3 6 0, 7 1 4 2].  
 The number of queens that are not attacking each other = 8.

SJ  
29/10/2024

Number of queen = 8

600

Number of queen = 8

Number of queen = 8

Code:

```
!sed -i 's/from joblib.my_exceptions import WorkerInterrupt/import joblib/'  
/usr/local/lib/python3.10/dist-packages/mlrose_hiive/runners/_nn_runner_base.py  
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[i] != position[j]) and (position[i] != position[j] + (j - i)) and (position[i] !=  
                position[j] - (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])  
  
best_position, best_objective, _ = mlrose.simulated_annealing(  
    problem=problem, schedule=T, max_attempts=500, max_iters=5000, 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 6 1 3 7 0 4]  
The number of queens that are not attacking each other is: 8.0
```

## Program 6

Implement PROPORTIONAL ENTAILMENT ALGORITHM

Algorithm:

12/9/24 LAB-7

URBAN  
EDGE

### Proportional Entailment Algorithm

function  $\text{PT-ENTAILS?}(KB, \alpha)$  returns true or false

Inputs:  $KB$ , The knowledge base, a sentence in propositional logic

$\alpha$ , The query, a sentence in propositional logic

Symbols  $\leftarrow$  a list of the propositional symbols in  $KB \cup \alpha$ .

return  $\text{TT-check-all}(KB, \alpha, \text{symbols}, \text{es})$

function  $\text{TT-check-all}(KB, \alpha, \text{symbols}, \text{model})$  returns true or false

if ( $\text{empty?}(\text{symbols})$ ) then

  if ( $\text{PL-True?}(KB, \text{model})$ ) then

    return  $\text{PL-True?}(\alpha, \text{model})$

  else return true

else do

$p \in \text{First}(\text{symbols})$

  rest  $\leftarrow \text{Rest}(\text{symbols})$

  return  $\text{TT-check-all}(KB, \alpha, \text{rest}, \text{model} \cup \{p = \text{true}\})$

and

~~$\text{TT-check-all}(KB, \alpha, \text{rest}, \text{model} \cup \{p = \text{false}\})$~~

propositional Inference : Enumeration Method

(cont. 197) part 1: inference

$$2 = A \vee B \quad \vee B = (A \vee C) \wedge (B \vee \neg C)$$

Checking that  $KD \vdash 2$

A	B	C	$A \vee C$	$B \vee \neg C$	$KD$	2
false	f	f	f	t	f	f
f	f	t	t	t	f	f
f	t	f	t	t	t	
f	f	f	t	t	t	
t	f	t	t	t	t	
t	f	f	t	t	t	
t	t	f	t	t	t	
t	t	t	t	t	t	

Output:  $\{(\text{'And'}), (\text{'!OR'}, \text{'A'}, \text{'B'})\}, (\text{'!OR'}, \text{'not'}, \text{'C'})\}, (\text{'!B'})\}$   
 $\{(\text{'OR'}, \text{'A'}, \text{'B'})\}$

Does KB extract walpha? True

14/11/2024

Value of  $\neg \neg A$  (T)

Code:

```
import itertools

# Function to evaluate an expression
def evaluate_expression(a, b, c, expression):
    # Use eval() to evaluate the logical expression
    return eval(expression)

# Function to generate the truth table and evaluate a logical expression
def truth_table_and_evaluation(kb, query):
    # All possible combinations of truth values for a, b, and c
    truth_values = [True, False]
    combinations = list(itertools.product(truth_values, repeat=3))
    # Reverse the combinations to start from the bottom (False -> True)
    combinations.reverse()

    # Header for the full truth table
    print(f'{a}:<5} {b}:<5} {c}:<5} {KB}:<20} {Query}:<20}')
    print("-" * 50)

    # Evaluate the expressions for each combination
    for combination in combinations:
        a, b, c = combination

        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)

        # Convert boolean values of a, b, c, KB, and Query to strings "True"/"False"
        a_str = "True" if a else "False"
        b_str = "True" if b else "False"
        c_str = "True" if c else "False"
        kb_result_str = "True" if kb_result else "False"
        query_result_str = "True" if query_result else "False"

        # Print the results for the knowledge base and the query
        print(f'{a_str}:<5} {b_str}:<5} {c_str}:<5} {kb_result_str}:<20} {query_result_str}:<20}')

    # Additional output for combinations where both KB and Query are true
    print("\nCombinations where both KB and Query are True:")
    print(f'{a}:<5} {b}:<5} {c}:<5} {KB}:<20} {Query}:<20}')
    print("-" * 50)

    # Print only the rows where both KB and Query are True
    for combination in combinations:
        a, b, c = combination
```

```

# Evaluate the knowledge base (KB) and query expressions
kb_result = evaluate_expression(a, b, c, kb)
query_result = evaluate_expression(a, b, c, query)

# If both KB and query are True, print the combination
if kb_result and query_result:
    a_str = "True" if a else "False"
    b_str = "True" if b else "False"
    c_str = "True" if c else "False"
    kb_result_str = "True" if kb_result else "False"
    query_result_str = "True" if query_result else "False"
    print(f'{a_str}<5} {b_str}<5} {c_str}<5} {kb_result_str}<20} {query_result_str}<20}")

# Define the logical expressions as strings
kb = "(a or c) and (b or not c)" # Knowledge Base
query = "a or b" # Query to evaluate

# Generate the truth table and evaluate the knowledge base and query
truth_table_and_evaluation(kb, query)

```

OUTPUT:

a	b	c	KB	Query
False	False	False	False	False
False	False	True	False	False
False	True	False	True	True
False	True	True	True	True
True	False	False	False	True
True	False	True	False	True
True	True	False	True	True
True	True	True	True	True

Combinations where both KB and Query are True:

a	b	c	KB	Query
False	True	False	True	True
False	True	True	True	True
True	True	False	True	True
True	True	True	True	True

## Program 7

Implement UNIFICATION ALGORITHM

Algorithm:

19/11/24	LAB - 8	URBAN EDGE
heading	Implementation of unification algorithm	
	Algorithm: Unify( $\psi_1, \psi_2$ )	
	$\psi_1, \psi_2$ are terms or predicates.	
1)	$\psi_1$ 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 false. c) Else if $\psi_2$ is a variable, a. If $\psi_2$ occurs as $\psi_1$ then, return failure False. d) Else return false.	
2)	If the two predicates have different symbols in $\psi_1$ and $\psi_2$ are not same, then return false.	
3)	If $\psi_1$ and $\psi_2$ have a different number of arguments, then return failure.	
4)	Set substitution set (subst) to NIL.	
5)	Return $\&$ subst.	

$$\text{Q1) } \psi_1 = P(b, x, f(g(z))) \quad -\textcircled{1}$$

$$\psi_2 = P(c^2, f(y), f(y)) \quad -\textcircled{2}$$

Replace  $Pn$   $\textcircled{2}$  with  $b$ ,

$$P(b, f(y), f(y)) \quad b \neq c^2$$

Replace  $f \in Pn$   $\textcircled{1}$  with  $f(y)$

$$= P(b, f(y), f(y))$$

$$= P(b, f(y), f(g(z))) \quad \text{using } f(y) \text{ in}$$

Replace  $g \in Pn$   $\textcircled{2}$  with  $g(z)$

$$= P(b, f(y), f(g(z))) \quad \text{using } g(z) \text{ in}$$

$$= P(b, f(y), f(g(z))) \quad -\textcircled{3}$$

$$= P(b, f(y), f(g(z))) = \textcircled{1}$$

Unification possible

$$\text{Q2) } \psi_1 = P(f(a), g(y))$$

$$\psi_2 = P(x, x)$$

Unification not possible as  $x$  cannot be replaced.

OUTPUT:

Q1) Unification successful

Substitution:  $\{('b'; 'z'), ('x'; ('f', 'y')), ('y'; ('g', 'z'))\}$

Q2) Output:

(A) -  $((\alpha \beta)^*, \delta, \text{dig} = 10)$

Unification Failed.

~~Predicate mismatch:  $(\alpha \beta, \text{dig})$~~

~~Predicate mismatch at end:  $\neq \text{dig}$~~

Sb  
19/11/24

tab 10 output:  $((\alpha \beta)^*, \delta, \text{dig})$

Don John likes Peanuts? Yes

~~Yello 91  
 $((\alpha \beta)^*, (\delta)^*, \text{dig})$~~

(A) -  $((\alpha \beta)^*, (\delta)^*, \text{dig}) =$

(A) -  $((\alpha \beta)^*, (\delta)^*, \text{dig}) =$

Code:

```
def unify(phi1, phi2, subst={}):
    print(f'Attempting to unify: {phi1} with {phi2}')
    print(f'Current substitution: {subst}')

    if isinstance(phi1, str) and phi1.islower():
        return unify_var(phi1, phi2, subst)
    if isinstance(phi2, str) and phi2.islower():
        return unify_var(phi2, phi1, subst)

    if phi1 == phi2:
        return subst
```

```

if isinstance(phi1, tuple) and isinstance(phi2, tuple):
    if phi1[0] != phi2[0]:
        print("Predicate mismatch:", phi1[0], "vs", phi2[0])
        return None

    if len(phi1) != len(phi2):
        print("Argument count mismatch")
        return None

    for arg1, arg2 in zip(phi1[1:], phi2[1:]):
        subst = unify(arg1, arg2, subst)
        if subst is None:
            return None
        return subst

    return None

def unify_var(var, value, subst):
    print(f'Unifying variable: {var} with value: {value}')

    if var in subst:
        return unify(subst[var], value, subst)
    elif value in subst:
        return unify(var, subst[value], subst)
    elif occurs_check(var, value, subst):
        print(f"Occurs check failed: {var} in {value}")
        return None
    else:
        subst[var] = value
        return subst

def occurs_check(var, value, subst):
    if var == value:
        return True
    elif isinstance(value, tuple):
        return any(occurs_check(var, arg, subst) for arg in value)
    elif value in subst:
        return occurs_check(var, subst[value], subst)
    return False

phi1 = ("q", "b", "x", ("f", ("g", "z")))
phi2 = ("q", "z", ("f", "y"), ("f", "y"))

result = unify(phi1, phi2)
if result:
    print("\nUnification Successful!")

```

```
    print("Substitution:", result)
else:
    print("\nUnification Failed!")
```

OUTPUT:

Attempting to unify: ('q', 'b', 'x', ('f, ('g', 'z'))) with ('q', 'z', ('f, 'y'), ('f, 'y'))

Current substitution: {}

Attempting to unify: b with z

Current substitution: {}

Unifying variable: b with value: z

Attempting to unify: x with ('f, 'y')

Current substitution: {'b': 'z'}

Unifying variable: x with value: ('f, 'y')

Attempting to unify: ('f, ('g', 'z')) with ('f, 'y')

Current substitution: {'b': 'z', 'x': ('f, 'y')}

Attempting to unify: ('g', 'z') with y

Current substitution: {'b': 'z', 'x': ('f, 'y')}

Unifying variable: y with value: ('g', 'z')

Unification Successful!

Substitution: {'b': 'z', 'x': ('f, 'y'), 'y': ('g', 'z')}

## Program 8

Implement FIRST ORDER LOGIC – FORWARD REASONING

Algorithm:

26/11/24	LAB - 9	URBAN EDGE
First Order Logic		
function FOL - FC ASK (KB12) returns		
in form a; subs in or if false return ()		
Inputs: i) KB P knowledges base, open (a set of first-order definite clause)		
ii) A query A is something like (a set of atoms)		
Q: - The query, an atomic structure		
1. Initialize new (empty) local variables, :27 new (for the new sentences)		
with repeat until new is empty		
(control loop)		
2. Process, without for each rule r in KB do		
i) Bind q to (p, M0 - o1 ∩ p1) ↪ q ↪		
standardized variables clause		
ii) Unify with q for each o such that SUBST (o, p, M0, ..., n) ↪ q ↪		
binding = SUBST (o, p, M0, ..., n)		
if binding fails then return false		
else add new to KB		
q' ↪ SUBST (o, q)		
if q' does not unify with some		
existing sentence already in KB or		
new then		
Add q' to new		
if c unify (q', d)		
if o ps. not fail. then return		
add new to KB		
return false		

```

Code:
# Define the knowledge base (KB) as a set of facts
KB = set()

# Premises based on the provided FOL problem
KB.add('American(Robert)')
KB.add('Enemy(America, A)')
KB.add('Missile(T1)')
KB.add('Owns(A, T1)')

# Define inference rules
def modus_ponens(fact1, fact2, conclusion):
    """Apply modus ponens inference rule: if fact1 and fact2 are true, then conclude conclusion."""
    if fact1 in KB and fact2 in KB:
        KB.add(conclusion)
        print(f"Inferred: {conclusion}")

def forward_chaining():
    """Perform forward chaining to infer new facts until no more inferences can be made."""
    # 1. Apply: Missile(x) → Weapon(x)
    if 'Missile(T1)' in KB:
        KB.add('Weapon(T1)')
        print(f"Inferred: Weapon(T1)")

    # 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1)
    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
        KB.add('Sells(Robert, T1, A)')
        print(f"Inferred: Sells(Robert, T1, A)")

    # 3. Apply: Hostile(A) from Enemy(A, America)
    if 'Enemy(America, A)' in KB:
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")

    # 4. Check if the goal is reached: Criminal(Robert)
    if (
        'American(Robert)' in KB and
        'Weapon(T1)' in KB and
        'Sells(Robert, T1, A)' in KB and
        'Hostile(A)' in KB
    ):
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")

    # Check if the conclusion has been reached
    if 'Criminal(Robert)' in KB:

```

```
    print("Conclusion: Robert is a criminal!")
else:
    print("No more inferences can be made.")

# Run forward chaining to attempt to derive the conclusion
forward_chaining()
```

OUTPUT:

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

## Program 9

Convert a given first order logic statement into Conjunctive Normal Form (CNF).

Algorithm:

3/14/29 LAB - 10 URBAN EDGE

Algorithm: FOL TO CNF

- 1) Convert All Sentences to CNF
- 2) Negate Conclusion & Convert result to CNF
- 3) Add conclusion S & convert result to CNF
- 4) Add Repeat until contradiction or no progress. If no contradiction is found:
  - a) Select 2 clauses (call them parent clauses)
  - b) Resolve them together, performing substitution until all required unifications are done.
  - c) If the resulting resolvent is the empty clause A contradiction has been found.
  - d) If not, add resolvent to the premises.
- 5) Since on step 4 process the conclusion.

(b, p) now  $\rightarrow$  p  
as of now b is not in the premises

Lab 9 output:

primed attack path : m1t1m1n1

Inferred = Wrap on CTI

→ Inferred : m1t1S1S1 (Robert), T1K1P1A1

Inferred : Hostile (A)

Inferred : Criminal (Robert)

( $\exists x \forall Robert \exists y \forall m1 \exists t1 \exists S1 \exists P1 \exists A1$ )

M1t1 (m1t1) m1t1n1 n1 with all m1s

in scope

Lab 10 output:

What: Does John like Peanuts? (Yes).

↓ below ↓ with a

Peanuts (P, x, likes) test = John likes Peanuts

(C100), P1101P1

$\neg x \rightarrow 20$

$\neg x$  (John)  $\neg$  likes (John)  $\neg$  likes (John)

( $\neg x$ )  $\neg$  likes (John)  $\neg$  likes (John)  $\neg$  likes (John)  $\neg$  likes (John)

(( $\neg x$ ), a)

V number with q <= 12

(V, x) x > 12

V number

A window (A, x, ends)  $\neg$  ends - 41M  $\neg$  ends - 41M

windows (41M)

current end 4081 test = John likes Peanuts

Code:

```
# Define the knowledge base (KB)
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 -11

URBAN  
EDGE

Algorithm: Alpha-beta Pruning

function ALPHA-BETA (state) returns an

(a) action : action

(b) value : utility

$v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$

return the action  $a$  in ACTIONS(state) with

value  $v$

: function at last

.function MAX-VALUE (state,  $\alpha$ ,  $\beta$ ) returns

a utility value

if TERMINAL-TEST (state,  $\alpha$ ,  $\beta$ ) returns :

UTILITY (state)

$v \leftarrow -\infty$

for each  $a$  in ACTIONS (state) do

$v \leftarrow \text{MAX} (v, \text{MIN-VALUE} (\text{RESULT}(s, a),$

$\alpha, \beta))$

if  $v \geq \beta$  then return  $v$

$\alpha \leftarrow \text{MAX} (\alpha, v)$

return  $v$

function MIN-VALUE (state,  $\alpha$ ,  $\beta$ ) returns a

utility value

if TERMINAL-TEST (state) then return

UTILIT V (state)

$v \in +\infty$

for each  $a$  in ACTIONS(state) do

$v \leftarrow \min(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$

If  $v \leq \alpha$  then return  $v$

$\beta \leftarrow \min(\beta, v)$

return  $v$

OUTPUT :

DO

Enter numbers for the game Tree  
(space-separated) ? 10 9 14  
18 5 4 50 3

Final Result of Alpha-beta  
Pruning : 50.

S  
3/12/2024

Code:

```
# Alpha-Beta Pruning Implementation
def alpha_beta_pruning(node, alpha, beta, maximizing_player):
    # Base case: If it's a leaf node, return its value (simulating evaluation of the node)
    if type(node) is int:
        return node

    # If not a leaf node, explore the children
    if maximizing_player:
        max_eval = -float('inf')
        for child in node: # Iterate over children of the maximizer node
            eval = alpha_beta_pruning(child, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval) # Maximize alpha
            if beta <= alpha: # Prune the branch
                break
        return max_eval
    else:
        min_eval = float('inf')
        for child in node: # Iterate over children of the minimizer node
            eval = alpha_beta_pruning(child, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval) # Minimize beta
            if beta <= alpha: # Prune the branch
                break
        return min_eval

# Function to build the tree from a list of numbers
def build_tree(numbers):
    # We need to build a tree with alternating levels of maximizers and minimizers
    # Start from the leaf nodes and work up
    current_level = [[n] for n in numbers]
    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            if i + 1 < len(current_level):
                next_level.append(current_level[i] + current_level[i + 1]) # Combine two nodes
            else:
                next_level.append(current_level[i]) # Odd number of elements, just carry forward
        current_level = next_level
    return current_level[0] # Return the root node, which is a maximizer

# Main function to run alpha-beta pruning
def main():
    # Input: User provides a list of numbers
    numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))
```

```
# Build the tree with the given numbers
tree = build_tree(numbers)

# Parameters: Tree, initial alpha, beta, and the root node is a maximizing player
alpha = -float('inf')
beta = float('inf')
maximizing_player = True # The root node is a maximizing player

# Perform alpha-beta pruning and get the final result
result = alpha_beta_pruning(tree, alpha, beta, maximizing_player)
print("Final Result of Alpha-Beta Pruning:", result)

# Run the main function
if __name__ == "__main__":
    main()
```

OUTPUT:

```
Enter numbers for the game tree (space-separated): 3 5 6 9 1 2 0 -1
Final Result of Alpha-Beta Pruning: 6
```