VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Vonteddu Karuneshwar Reddy (1BM22CS332)

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 **Vonteddu Karuneshwar Reddy (1BM22CS332) ,**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.

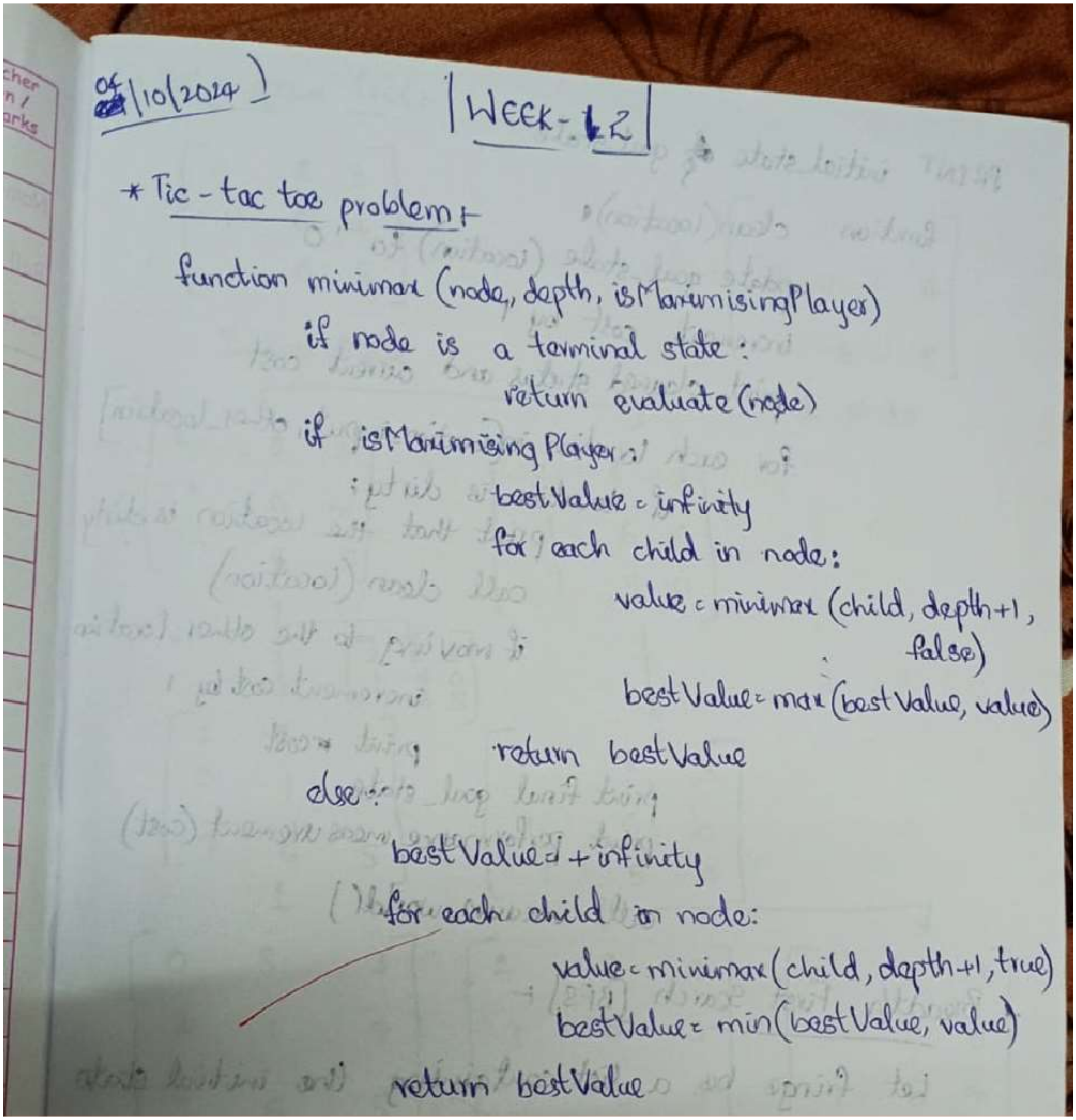
Swathi sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor& HOD Department of CSE, BMSCE

Index

GITHUB LINK: https://github.com/Karun04V/AI_1BM22CS332

Program 1
Implement Tic –Tac –Toe
Game

Algorithm:



04/10/2024

|WEEK-2|

* Tic-tac toe problem:-

function minimax (node, depth, isMaximisingPlayer)
    if node is a terminal state:
        return evaluate(node)
    if isMaximising Player:
        bestValue = infinity
        for each child in node:
            value = minimax (child, depth+1, false)
            bestValue = max (bestValue, value)
        return bestValue
    else:
        bestValue = +infinity
        for each child in node:
            value = minimax (child, depth+1, true)
            bestValue = min (bestValue, value)
    return bestValue

Code:

```
print("USN: 1BM22CS332")
board = {1: ' ', 2: ' ', 3: ' ',
         4: ' ', 5: ' ', 6: ' ',
         7: ' ', 8: ' ', 9: ' '}

def printBoard(board):
    print(board[1] + '|' + board[2] + '|' + board[3])
    print('-+-+-')
    print(board[4] + '|' + board[5] + '|' + board[6])
    print('-+-+-')
    print(board[7] + '|' + board[8] + '|' + board[9])
    print('\n')

def spaceFree(pos):
    return board[pos] == ' '

def checkWin():
    if (board[1] == board[2] == board[3] != ' ' or
        board[4] == board[5] == board[6] != ' ' or
        board[7] == board[8] == board[9] != ' ' or
        board[1] == board[5] == board[9] != ' ' or
        board[3] == board[5] == board[7] != ' ' or
        board[1] == board[4] == board[7] != ' ' or
        board[2] == board[5] == board[8] != ' ' or
        board[3] == board[6] == board[9] != ' '):
        return True
    return False

def checkMoveForWin(move):
    if (board[1] == board[2] == board[3] == move or
        board[4] == board[5] == board[6] == move or
        board[7] == board[8] == board[9] == move or
        board[1] == board[5] == board[9] == move or
        board[3] == board[5] == board[7] == move or
        board[1] == board[4] == board[7] == move or
        board[2] == board[5] == board[8] == move or
        board[3] == board[6] == board[9] == move):
        return True
    return False

def checkDraw():
    return all(space != ' ' for space in board.values())

def insertLetter(letter, position):
```

```
    if spaceFree(position):
        board[position] = letter
        printBoard(board)

        if checkDraw():
            print('Draw!')
            return "Game Over"
        elif checkWin():
            if letter == 'X':
                print('Bot wins!')
                return "Game Over"
            else:
                print('You win!')
                return "Game Over"
    else:
        print('Position taken, please pick a different position.')
        position = int(input('Enter new position: '))
        insertLetter(letter, position)

player = 'O'
bot = 'X'

def playerMove():
    position = int(input('Enter position for O: '))
    return insertLetter(player, position)

def compMove():
    bestScore = -1000
    bestMove = 0
    for key in board.keys():
        if board[key] == ' ':
            board[key] = bot
            score = minimax(board, False)
            board[key] = ' '
            if score > bestScore:
                bestScore = score
                bestMove = key

    result = insertLetter(bot, bestMove)
    if result == "Game Over":
        return "Game Over"

def minimax(board, isMaximizing):
    if checkMoveForWin(bot):
        return 1
    elif checkMoveForWin(player):
        return -1
```

```
    elif checkDraw():
        return 0

    if isMaximizing:
        bestScore = -1000

        for key in board.keys():
            if board[key] == ' ':
                board[key] = bot
                score = minimax(board, False)
                board[key] = ' '
                bestScore = max(score, bestScore)
        return bestScore
    else:
        bestScore = 1000

        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
                bestScore = min(score, bestScore)
        return bestScore

while True:
    if compMove() == "Game Over":
        break
    if playerMove() == "Game Over":
        Break
```

Output:

```
USN: 1BM22CS332
V KARUNESHWAR REDDY
X| |
-+-+-
 | |
-+-+-
 | |


Enter position for O:  5
X| |
-+-+-
 |O|
-+-+-
 | |


X|X|
-+-+-
 |O|
-+-+-
 | |


Enter position for O:  4
X|X|
-+-+-
O|O|
-+-+-
 | |


X|X|X
-+-+-
O|O|
-+-+-
 | |


Bot wins!
```

Implement Vaccum Cleaner Agent

18/10/2024 WEEK-3

* Vacuum Cleaner:

function vacuum_world()
initialize goal_state as ("A": "0", "B": "0")
initialize cost as 0
get location_input from user
get status_input for location_input from user
get other_location based on location_input
get status_input_compliment for other_location from user

PRINT initial_state & goal_state

function clean(location)
update goal_state (location) to '0'
increment cost by 1
print cleaned status and current cost

for each location in [location_input, other_location]
if location is dirty:
print that the location is dirty
call clean (location)
if moving to the other_location
increment cost by 1
print cost
print final goal_state
print performance measurement (cost)

call vacuum_world()

Code:

```
print("USN: 1BM22CS332")
print("V KARUNESHWAR REDDY")
class VacuumCleaner:
    def __init__(self):
        # Initialize places A and B as either 'Dirty' or 'Clean'
        self.places = {'A': 'Dirty', 'B': 'Dirty'}
        # Start the vacuum cleaner at place A
        self.current_position = 'A'

    def check(self):
        # Check if the current position is dirty
        if self.places[self.current_position] == 'Dirty':
            print(f"Place {self.current_position} is Dirty.")
            return True
        else:
            print(f"Place {self.current_position} is Clean.")
            return False

    def suck(self):
        # Clean the current position if it's dirty
        if self.check():
            print(f"Cleaning place {self.current_position}.")
            self.places[self.current_position] = 'Clean'
        else:
            print(f"Place {self.current_position} is already clean.")

    def move(self):
        # Move to the other place
        self.current_position = 'B' if self.current_position == 'A' else 'A'
        print(f"Moving to place {self.current_position}.")

    def start_cleaning(self):
        # Start the cleaning process
        for _ in range(2):  # Loop twice to cover both places
            self.suck()     # Clean the current position if dirty
            self.move()     # Move to the other position

# Create a vacuum cleaner instance
vacuum = VacuumCleaner()

# Start the cleaning process
vacuum.start_cleaning()
```

Output:

```
USN: 1BM22CS332
V KARUNESHWAR REDDY
Place A is Dirty.
Cleaning place A.
Moving to place B.
Place B is Dirty.
Cleaning place B.
Moving to place A.
```

Program 2

Implement 8 puzzle problems using (DFS) and (BFS)

* DFS :-

Loop :
  if fringe is empty return failure
  Node ← remove_first (fringe)
  if Node is a goal
    then return the path from initial state to Node
  else
    generate all successors of Node and add generated nodes to the front of fringe.
End Loop

→ State space :-

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 0 & 7 & 8 \end{bmatrix} \text{(initial state)}$$

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

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

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

CODE:for dfs

Input:

```
print("1BM22CS
332")
print("V
KARUNESHWA
R REDDY")
from collections
import deque

def
solve_8puzzle_df
s(initial_state):
    """
    Solves the 8-
puzzle using
Depth-First
Search.

    Args:
        initial_state:
A list of lists
representing the
initial state of the
puzzle.

    Returns:
        A list of lists
representing the
solution path, or
None if no
solution is found.
    """

    def
find_blank(state):
        """Finds the
row and column
of the blank tile
(0)."""
        for row in
range(3):
```

```
            for col in range(3):
                if state[row][col] == 0:
                    return row, col

    def get_neighbors(state):
        """Generates possible neighbor states by moving the blank tile."""
        row, col = find_blank(state)
        neighbors = []
        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_state = [r[:] for r in state]
                new_state[row][col], new_state[new_row][new_col] = new_state[new_row][new_col], new_state[row][col]
```

```
neighbors.append(
new_state)

        return
neighbors

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

    # Print initial
and goal states
    print("Initial
State:")
    for row in
initial_state:
        print(row)
    print("\nGoal
State:")
    for row in
goal_state:
        print(row)

print("\nStarting
DFS...\n")

    stack =
[(initial_state, [])]
    visited = set()

    while stack:
        current_state,
path = stack.pop()

        # Convert
state to tuple for
easy set
comparison
        state_tuple =
tuple(map(tuple,
current_state))
```

```
        # Skip if
already visited
        if state_tuple
in visited:
            continue

        # Mark as
visited

visited.add(state_t
uple)

        # Check if
the goal state is
reached
        if
current_state ==
goal_state:
            return path
+ [current_state]

        # Explore
neighbors
        for neighbor
in
get_neighbors(cur
rent_state):

stack.append((nei
ghbor, path +
[current_state]))

    return None  #
No solution found
# Example usage:
initial_state = [[1,
2, 3], [4, 5, 6], [0,
7, 8]]
solution =
solve_8puzzle_df
s(initial_state)
if solution:
```

```
print("\nSolution found:")
    for state in solution:
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

Output:

```
1BM22CS332
V KARUNESHWAR REDDY
Initial State:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

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

Starting DFS...
```

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

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

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

BFS:

* Breadth First Search [BFS] :-

Let fringe be a list containing the initial state

```
Loop    if fringe is empty return failure
        Node ← remove_first (fringe)
        if Node is a goal
                return the path from initial
                    state to Node
        else
                generate all successors of Node and
                add generated nodes to the back of
                fringe.
End loop
```

→ State space tree

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

Initial state — End state

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

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

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

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

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

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

CODE: for bfs

```
print("1BM22CS332")
print("V KARUNESHWAR REDDY")
from collections import deque

def solve_8puzzle_bfs(initial_state):
    """
    Solves the 8-puzzle using Breadth-First Search.

    Args:
        initial_state: A list of lists representing the initial state of the puzzle.

    Returns:
        A list of lists representing the solution path, or None if no solution is found.
    """

    def find_blank(state):
        """Finds the row and column of the blank tile (0)."""
        for row in range(3):
            for col in range(3):
                if state[row][col] == 0:
                    return row, col

    def get_neighbors(state):
        """Generates possible neighbor states by moving the blank tile."""
        row, col = find_blank(state)
        neighbors = []

        # Possible moves: Up, Down, Left, Right
        if row > 0:  # Up
            new_state = [r[:] for r in state]
            new_state[row][col], new_state[row - 1][col] = new_state[row - 1][col], new_state[row][col]
            neighbors.append(new_state)
        if row < 2:  # Down
            new_state = [r[:] for r in state]
            new_state[row][col], new_state[row + 1][col] = new_state[row + 1][col], new_state[row][col]
            neighbors.append(new_state)
        if col > 0:  # Left
            new_state = [r[:] for r in state]
            new_state[row][col], new_state[row][col - 1] = new_state[row][col - 1], new_state[row][col]
            neighbors.append(new_state)
        if col < 2:  # Right
            new_state = [r[:] for r in state]
            new_state[row][col], new_state[row][col + 1] = new_state[row][col + 1], new_state[row][col]
            neighbors.append(new_state)
```

```
        return neighbors

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

    # Print initial and goal states
    print("Initial State:")
    for row in initial_state:
        print(row)
    print("\nGoal State:")
    for row in goal_state:
        print(row)
    print("\nStarting BFS...\n")

    queue = deque([(initial_state, [])])
    visited = set()

    while queue:
        current_state, path = queue.popleft()

        # Check if the goal state is reached
        if current_state == goal_state:
            return path + [current_state]

        # Mark the current state as visited
        visited.add(tuple(map(tuple, current_state)))

        # Explore neighbors
        for neighbor in get_neighbors(current_state):
            if tuple(map(tuple, neighbor)) not in visited:
                queue.append((neighbor, path + [current_state]))

    return None  # No solution found

# Example usage:
initial_state = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]
solution = solve_8puzzle_bfs(initial_state)
if solution:
    print("\nSolution found:")
    for state in solution:
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

Output:

```
1BM22CS332
V KARUNESHWAR REDDY
Initial State:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

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

Starting BFS...
```

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

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

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

Program 3

Implement A* Search Algorithm

25/10/2024

WEEK-03

A* Algorithm:-

```
function A* search (problem) returns a solution or failure
    node ← a node n with n.state = problem.initial_state,
                                    n.g = 0
    frontier ← a priority queue ordered by ascending
               g+h, only element n
    loop do
        if empty? (frontier) then return failure
        n ← pop (frontier)
        if problem.goalTest (n.state) then return solution (n)
        for each action a in problem.actions (n.state) do
            n' ← childNode (problem, n, a)
            insert(n', g(n') + h(n'), frontier)
```

→ $f(n) = g(n) + h(n)$ where

$f(n)$ is the evaluation function which gives cheapest solution cost,

$g(n)$ is the cost to reach that node from initial state (i.e, depth)

$h(n)$ is an estimation of assumed cost from current state to reach the goal state.

1) $h(n) \Rightarrow$ No. of misplaced tiles
2) $h(n) \Rightarrow$ Manhattan Distance.

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & 0 & 5 \end{bmatrix} \quad g(n)=0,\ h(n)=4 \Rightarrow \boxed{f(n)=4} \quad \begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

(initial) (goal)

$g(n)=1$

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

$h(n)=3 \Rightarrow f(n)=4$ | $h(n)=5$, $f(n)=6$ | $h(n)=5$, $f(n)=6$

$g(n)=2$

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

$h(n)=3 \Rightarrow f(n)=5$ | $h(n)=3$, $f(n)=5$ | $h(n)=4$, $f(n)=6$

$g(n)=3$

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

$h(n)=2 \Rightarrow f(n)=5$ | $h(n)=4$, $f(n)=7$

$g(n)=4$

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix} \quad h(n)=1 \Rightarrow f(n)=5 \longrightarrow$$

$g(n)=5$

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix} \quad h(n)=0,\ \boxed{f(n)=5}$$

2) Manhattan Distance :- $h(n)$ = Manhattan Distance

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & 0 & 5 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix} \text{ (goal)}$$

(initial)

$g(n)=0$

$h(n)=1+2+0+1+1+0+0+0$

$\Rightarrow h(n)=5 \Rightarrow \boxed{f(n)=5}$

$g(n)=1$

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

$h(n)=1+2+0+1+0+0+0+0 \Rightarrow h(n)=4 \Rightarrow \boxed{f(n)=5}$

$h(n)=1+2+0+1+1+0+0+1 \Rightarrow h(n)=6$, $\boxed{f(n)=7}$

$h(n)=1+2+0+1+1+0+1+0 \Rightarrow h(n)=6$, $\boxed{f(n)=7}$

$g(n)=2$

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

$h(n)=1+0+1+1+0+0+0+0 \Rightarrow h(n)=3 \Rightarrow f(n)=5$

$h(n)=1+1+0+2+0+0+0+0 \Rightarrow h(n)=4$, $f(n)=6$

$h(n)=1+2+0+1+1+0+0+0 \Rightarrow \boxed{h(n)=5}$, $f(n)=7$

45/10

$g(n)=3$

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

$h(n)=1+0+0+1+1+0+0+0+0$, $\boxed{f(n)=5}$

$h(n)=1+1+1+1+0+0+0+0 \Rightarrow h(n)=4$, $f(n)=7$

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

$g(n)=4$, $h(n)=1+0+0+0+0+0+0+0 \Rightarrow \boxed{f(n)=5}$

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

$g(n)=5$, $h(n)=0$, $f(n)=5$

CODE:

MANHATTAN DISTANCE

```
import heapq
    def manhattan_distance(state, goal):
    distance = 0

    for i in range(3):

      for j in range(3):

        tile = state[i][j]

        if tile != 0:

          for r in range(3):

            for c in range(3):

              if goal[r][c] == tile:

                target_row, target_col = r, c

                break

          distance += abs(target_row - i) + abs(target_col - j)

    return distance

def findmin(open_list, goal):

minv = float('inf') best_state =

None

    for state in open_list:

      h = manhattan_distance(state['state'], goal)

      f = state['g'] + h

      if f < minv:

        minv = f

        best_state = state
```

```
    open_list.remove(best_state)
    return best_state


def operation(state):
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states


def find_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None
```

```
def apply_move(state, blank_pos, move):

    i, j = blank_pos

    new_state = [row[:] for row in state]

    if move == 'up' and i > 0:

        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]

    elif move == 'down' and i < 2:

        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]

    elif move == 'left' and j > 0:

        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]

    elif move == 'right' and j < 2:

        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]

    else:

        return None

    return new_state


def print_state(state):

    for row in state:

        print(' '.join(map(str, row)))

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

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


open_list = [{'state': initial_state, 'parent': None, 'move': None, 'g': 0}]

visited_states = []
```

```
while open_list:
    best_state = findmin(open_list, goal_state)

    h = manhattan_distance(best_state['state'], goal_state)
    f = best_state['g'] + h

    print(f"g(n) = {best_state['g']}, h(n) = {h}, f(n) = {f}")
    print_state(best_state['state'])
    print()

    if h == 0:
        print("Goal state reached!")
        break

    visited_states.append(best_state['state'])
    next_states = operation(best_state)

    for state in next_states:
        if state['state'] not in visited_states:
            open_list.append(state)

if h == 0:
```

```
        moves = []
        goal_state_reached = best_state
        while goal_state_reached['move'] is not None:
            moves.append(goal_state_reached['move'])
            goal_state_reached = goal_state_reached['parent']
        moves.reverse()
        print("\nMoves to reach the goal state:", moves)
    else:
        print("No solution found.")
```

```
g(n) = 0, h(n) = 5, f(n) = 5
2 8 3
1 6 4
7 0 5

g(n) = 1, h(n) = 4, f(n) = 5
2 8 3
1 0 4
7 6 5

g(n) = 2, h(n) = 3, f(n) = 5
2 0 3
1 8 4
7 6 5

g(n) = 3, h(n) = 2, f(n) = 5
0 2 3
1 8 4
7 6 5

g(n) = 4, h(n) = 1, f(n) = 5
1 2 3
0 8 4
7 6 5

g(n) = 5, h(n) = 0, f(n) = 5
1 2 3
8 0 4
7 6 5

Goal state reached!

Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']
```

Misplaced Tiles:

```
    import heapq
defind_blank_tile(st
    ate):
for i in range(3):
    for j in range(3):
if state[i][j] == 0:
    return i, j
    return None

def count_misplaced_tiles(state, goal):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal[i][j]:
                misplaced += 1
    return misplaced
def generate_moves(state):
    moves = []
    x, y = find_blank_tile(state)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
```

```
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
            moves.append(new_state)
    return moves

def print_state(state):
    for row in state:
        print(row)
    print()

def a_star_8_puzzle(start, goal):

    open_list = []

    heapq.heappush(open_list, (count_misplaced_tiles(start, goal), 0, start, None))

    visited = set()

    while open_list:

        f_n, g_n, current_state, previous_state = heapq.heappop(open_list)

        print(f"g(n) = {g_n}, h(n) = {f_n - g_n}, f(n) = {f_n}")
        print_state(current_state)
```

```
    if current_state == goal:
        print("Goal state reached!")
        return
    visited.add(tuple(map(tuple, current_state)))
    for move in generate_moves(current_state):
        move_tuple = tuple(map(tuple, move))
        if move_tuple not in visited:

            g_move = g_n + 1
            h_move = count_misplaced_tiles(move, goal)
            f_move = g_move + h_move
            heapq.heappush(open_list, (f_move, g_move, move, current_state))


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


a_star_8_puzzle(start_state, goal_state)
```

Output:

```
g(n) = 0, h(n) = 4, f(n) = 4
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

g(n) = 1, h(n) = 3, f(n) = 4
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

g(n) = 2, h(n) = 3, f(n) = 5
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

g(n) = 2, h(n) = 3, f(n) = 5
[2, 8, 3]
[0, 1, 4]
[7, 6, 5]

g(n) = 3, h(n) = 2, f(n) = 5
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

g(n) = 4, h(n) = 1, f(n) = 5
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

g(n) = 5, h(n) = 0, f(n) = 5
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

Goal state reached!
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem.

08-11-2024

WEEK - 05

Q: Implement Hill Climbing Algorithm to solve N-Queens Problem.

Algo:

N-Queens Problem: In this problem, we need to arrange given no. of Queens 'n' in an nxn grid such that no queen should kill each other [same row, same column, diagonally]

function HillClimbing(problem) returns a state that is local maximum

current ← makeNode [problem. INITIAL STATE]

loop do

neighbour ← A highest value successor of current ~~if neighbour~~

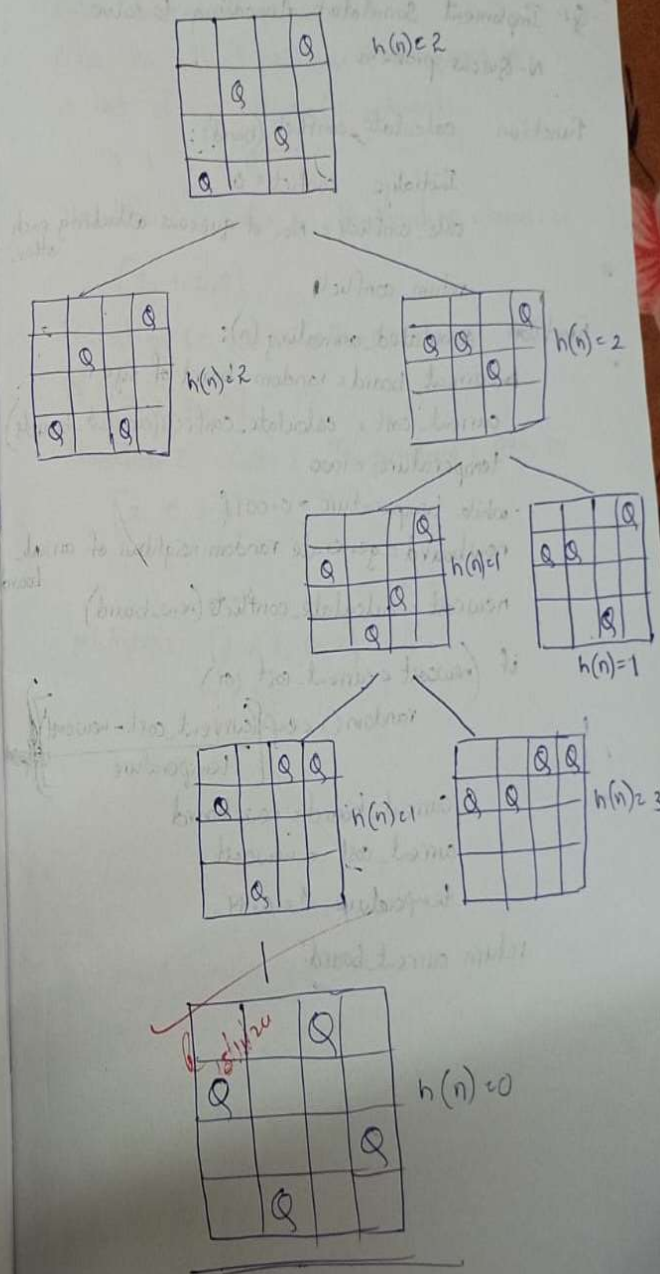
if neighbour : VALUE ≤ current. value then return current_state

current_neighbour

Steps:

1) Initial state
2) Evaluation fn
3) Generate Neighbour
4) Select Best Solution
5) Move to next neighbour
6) Repeat

State space tree:



```
import random
class NQueens:
    def _init_(self, n):
        self.n = n
        self.board = self.init_board()
    def init_board(self):
        # Randomly place one queen in each column
        return [random.randint(0, self.n - 1) for _ in range(self.n)]

    def fitness(self, board):
        # Count the number of pairs of queens attacking each other
        conflicts = 0
        for col in range(self.n):
            for other_col in range(col + 1, self.n):
                if board[col] == board[other_col] or abs(board[col] - board[other_col]) == abs(col - 
other_col):
                    conflicts += 1
        return conflicts
    def get_neighbors(self, board):
        neighbors = []
        for col in range(self.n):
            for row in range(self.n):
                if row != board[col]:  # Move queen to a different row in the same column
                    new_board = board[:]
```

```
            new_board[col] = row
            neighbors.append(new_board)
    return neighbors
def hill_climbing(self):
    current_board = self.board
    current_fitness = self.fitness(current_board)

    while current_fitness > 0:
        neighbors = self.get_neighbors(current_board)
        next_board = None
        next_fitness = current_fitness
        for neighbor in neighbors:
            neighbor_fitness = self.fitness(neighbor)
            if neighbor_fitness < next_fitness:
                next_fitness = neighbor_fitness
                next_board = neighbor
        if next_board is None:
            # Stuck at local maximum, can either return or restart
            print("Stuck at local maximum. Restarting...")
            self.board = self.init_board()
            current_board = self.board
            current_fitness = self.fitness(current_board)
        else:
```

```
                current_board = next_board
                current_fitness = next_fitness
        return current_board
# Example usage
if __name__ == "__main__":
    n = 4  # Size of the board (N)
    n_queens_solver = NQueens(n)
    solution = n_queens_solver.hill_climbing()

    print("Solution:")
    for row in solution:
        line = ['Q' if i == row else '.' for i in range(n)]
        print(' '.join(line))
```

Output:

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

Program 5

Simulated Annealing to Solve 8-Queens problem.

15/11/20..

WEEK-06

Q:- Implement Simulated Annealing to solve N-Queens problem.

```
Function calculate_conflicts(board):
    Initialize confict = 0
    calc_conflicts = No. of queeens attacking each other.
    return conflicts

Function simulated_annealing(n):
    current_board = random board of size n
    current_cost = calculate_conficts(current_boards)
    temperature = 1000
    while temperature > 0.001{
    newboard = generate random neighbour of current_board
    newcost = calculate_conflicts(new_board)
    if (newcost < current_cost (or)
            random() < exp[(current_cost-newcost)/temperature]
        current_board = new_Board
        current_cost = newcost.
        temperature *= 0.99
    return current_board
```

output:-

```
Enter the no of queens : 4
Enter the initial positions of the queens as
a list of row indices (0-indexed):
  3 1 2 0
Iteration 0:  Cost = 3, Temperature = 1000.00
  [2, 1, 2, 0]
Iteration 1: Cost = 3, Temperature = 990.00
  [2, 1, 2, 0]
Iteration 2: Cost = 2, Temperature = 980.10
  [2, 0, 2, 0]
  :
  :
solution :: [1, 3, 0, 2]
```

```
import random

import math


def print_board(state):

    size = len(state)

    for i in range(size):

        row = ['.'] * size

        row[state[i]] = 'Q'

        print(' '.join(row))

    print()


def calculate_conflicts(state):

    conflicts = 0

    size = len(state)

    for i in range(size):

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

            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):

                conflicts += 1

    return conflicts


def random_state(size):

    return [random.randint(0, size - 1) for _ in range(size)]
```

```
def neighbor(state):
    new_state = state[:]
    idx = random.randint(0, len(state) - 1)
    new_state[idx] = random.randint(0, len(state) - 1)
    return new_state


def simulated_annealing(size, initial_temp, cooling_rate):
    current_state = random_state(size)
    current_conflicts = calculate_conflicts(current_state)
    temperature = initial_temp

    while temperature > 1:
        new_state = neighbor(current_state)
        new_conflicts = calculate_conflicts(new_state)

        # If new state is better, accept it
        if new_conflicts < current_conflicts:
            current_state, current_conflicts = new_state, new_conflicts
        else:
            # Accept with a probability based on temperature
            acceptance_probability = math.exp((current_conflicts - new_conflicts) / temperature)
            if random.random() < acceptance_probability:
                current_state, current_conflicts = new_state, new_conflicts
```

```
        temperature *= cooling_rate

    return current_state

def main():
    size = 8
    initial_temp = 1000
    cooling_rate = 0.995

    solution = simulated_annealing(size, initial_temp, cooling_rate)
    print("Solution found:")
    print_board(solution)
    print("Conflicts:", calculate_conflicts(solution))

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

Output:

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

Conflicts: 6
```

Program 6:

29/11/2024

WEEK-12

Q: Creating a knowledge Base using propositional logic and proving query using resolution.

Initialize knowledge_base with propositional logic statements

Input Query:

Convert knowledge_base and query into CNF.

Add ¬query to CNF_clauses.

while True:

Select two clauses from CNF_clauses.

Resolve the clauses to produce a new clause

If new_clause is empty:

print("Query is proven using resolution")

break

If new_clause is not already in CNF_clauses:

Add new_clause to CNF_clauses

If no new_clause can be generated:

print("Query can't be proven using resolution")

break

20/12/24

O/p:

For knowledge_base = ["A", "B", "A∧B⇒C", "C⇒D"]

Query = "D"

Query is proven using resolution

```
def truth_table_entailment():
    print(f"{'A':<7}{'B':<7}{'C':<7}{'A or C':<12}{'B or not C':<15}{'KB':<8}{'alpha':<10}")
    print("-" * 65)
    all_entail = True
    for A in [False, True]:
      for B in [False, True]:
        for C in [False, True]:
          # Calculate individual components
          A_or_C = A or C                 # A or C
          B_or_not_C = B or (not C)       # B or not C
          KB = A_or_C and B_or_not_C      # KB = (A or C) and (B or not C)
          alpha = A or B                  # alpha = A or B

          # Determine if KB entails alpha for this row
          kb_entails_alpha = (not KB) or alpha  # True if KB implies alpha

          # If in any row KB does not entail alpha, set flag to False
          if not kb_entails_alpha:
            all_entail = False

          # Print the results for this row

   print(f"{str(A):<7}{str(B):<7}{str(C):<7}{str(A_or_C):<12}{str(B_or_not_C):<15}{str(KB):<8
   }{str(alpha):<10}")
```

```
    # Final result based on all rows
    if all_entail:
        print("\nKB entails alpha for all cases.")
    else:
        print("\nKB does not entail alpha for all cases.")


# Run the function to display the truth table and final result
truth_table_entailment()
```

Output:

```
A      B      C      A or C     B or not C    KB     alpha
--------------------------------------------------------------
False  False  False  False      True          False  False
False  False  True   True       False         False  False
False  True   False  False      True          False  True
False  True   True   True       True          True   True
True   False  False  True       True          True   True
True   False  True   True       False         False  True
True   True   False  True       True          True   True
True   True   True   True       True          True   True

KB entails alpha for all cases.
```

Program 7

Implement unification in first order logic.

22/11/2024

|WEEK-07|

Q: Implement unification in First Order Logic.

Algorithm :-

function Unify(Ψ_1, Ψ_2):

1) If Ψ_1 & Ψ_2 is a variable (or) constant, then,
   a) If Ψ_1 (or) Ψ_2 are identical, then return NIL.
   b) else if Ψ_1 is a variable,
      i) then if Ψ_1 occurs in Ψ_2, then return Failure.
      ii) else return $\{(\Psi_2/\Psi_1)\}$
   c) else if Ψ_2 is a variable
      i) If Ψ_2 occurs in Ψ_1, then return Failure.
      ii) else return $\{(\Psi_1/\Psi_2)\}$
   d) Else return Failure
2) If initial predicate symbol in Ψ_1 and Ψ_2 are not same, then return Failure.
3) If Ψ_1 and Ψ_2 have different no. of arguments, then return Failure.
4) Set substitution set (SUBST) to NIL.
5) For i=1 to no. of elements in Ψ_2
   a) Call Unity function with the i^{th} element of Ψ_1 and i^{th} element of Ψ_2 and put result into S.
   b) If S=failure, then returns Failure.
   c) If S≠NIL, then do,
      a) Apply S to the remainder of both L_1 & L_2.
      b) SUBST= APPEND (S, SUBST)
6) Return SUBST.

O/P:

expression a= "Eats (x, Apples)"
expression b= "Eats (Riya, y)"
} I/P

Unification successful

{'x': 'Riya', 'y': 'Apples'}

```
Perform unification on two expressions in first-order logic.

Args:
    expr1: The first expression (can be a variable, constant, or list representing a function).
    expr2: The second expression.
    substitution: The current substitution (dictionary).

Returns:
    A dictionary representing the most general unifier (MGU), or None if unification fails.
"""
if substitution is None:
    substitution = {}

# Debug: Print inputs and current substitution
print(f"Unifying {expr1} and {expr2} with substitution {substitution}")

# Apply existing substitutions to both expressions
expr1 = apply_substitution(expr1, substitution)
expr2 = apply_substitution(expr2, substitution)

# Debug: Print expressions after applying substitution
print(f"After substitution: {expr1} and {expr2}")
```

```
# Case 1: If expressions are identical, no substitution is needed
if expr1 == expr2:
    return substitution

# Case 2: If expr1 is a variable
if is_variable(expr1):
    return unify_variable(expr1, expr2, substitution)

# Case 3: If expr2 is a variable
if is_variable(expr2):
    return unify_variable(expr2, expr1, substitution)

# Case 4: If both are compound expressions (e.g., functions or predicates)
if is_compound(expr1) and is_compound(expr2):
    if expr1[0] != expr2[0] or len(expr1) != len(expr2):
        print(f"Failure: Predicate names or arity mismatch {expr1[0]} != {expr2[0]}")
        return None  # Function names or arity mismatch
    for arg1, arg2 in zip(expr1[1:], expr2[1:]):
        substitution = unify(arg1, arg2, substitution)
        if substitution is None:
            print(f"Failure: Could not unify arguments {arg1} and {arg2}")
            return None
```

```
        return substitution

    # Case 5: Otherwise, unification fails
    print(f"Failure: Could not unify {expr1} and {expr2}")
    return None

def unify_variable(var, expr, substitution):
    """
    Handles the unification of a variable with an expression.

    Args:
        var: The variable.
        expr: The expression to unify with.
        substitution: The current substitution.

    Returns:
        The updated substitution, or None if unification fails.
    """
    if var in substitution:
        # Apply substitution recursively
        return unify(substitution[var], expr, substitution)
    elif occurs_check(var, expr):
        # Occurs check fails if the variable appears in the term it's being unified with
```

```
        print(f"Occurs check failed: {var} in {expr}")
        return None
    else:
        substitution[var] = expr
        print(f"Substitution added: {var} -> {expr}")
        return substitution


def occurs_check(var, expr):
    """
    Checks if a variable occurs in an expression (to prevent cyclic substitutions).


    Args:
        var: The variable to check.
        expr: The expression to check against.


    Returns:
        True if the variable occurs in the expression, otherwise False.
    """
    if var == expr:
        return True
    elif is_compound(expr):
        return any(occurs_check(var, arg) for arg in expr[1:])
    return False
```

```
def is_variable(expr):
    """Checks if the expression is a variable."""
    return isinstance(expr, str) and expr[0].islower()


def is_compound(expr):
    """Checks if the expression is compound (e.g., function or predicate)."""
    return isinstance(expr, list) and len(expr) > 0


def apply_substitution(expr, substitution):
    """
    Applies a substitution to an expression.

    Args:
        expr: The expression to apply the substitution to.
        substitution: The current substitution.

    Returns:
        The updated expression with substitutions applied.
    """
    if is_variable(expr) and expr in substitution:
        return apply_substitution(substitution[expr], substitution)
    elif is_compound(expr):
```

```
        return [apply_substitution(arg, substitution) for arg in expr]
    return expr


# Example Usage:
expr1 = ['P', 'X', 'Y']
expr2 = ['P', 'a', 'Z']
result = unify(expr1, expr2)
print("Unification Result:", result)
```

Output:

```
Unifying ['P', 'X', 'Y'] and ['P', 'a', 'Z'] with substitution {}
After substitution: ['P', 'X', 'Y'] and ['P', 'a', 'Z']
Unifying X and a with substitution {}
After substitution: X and a
Substitution added: a -> X
Unifying Y and Z with substitution {'a': 'X'}
After substitution: Y and Z
Failure: Could not unify Y and Z
Failure: Could not unify arguments Y and Z
Unification Result: None
```

Program 8

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

29/11/2024

WEEK-08

Q: Implement Forward Reasoning Algorithm

function FOL_FC_ASK(KB, α) returns a substitution or false

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

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

repeat until new is empty

new ← { }

for each rule in KB do

$[(p_1 \wedge \ldots \wedge p_n) \Rightarrow q]$ ← STANDARDIZE_VARS(rule)

for each θ such that $[SUBST(\theta, p_1 \wedge \ldots \wedge p_n) = SUBST(\theta, p_1' \wedge \ldots \wedge p_n')]$ for some $p_1', \ldots, p_n'$ in KB

q' ← SUBST(θ, q)

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

add q' to new

ϕ ← UNIFY(q', α)

if ϕ is not fail then return ϕ

add new to KB

return false

o/p: Rule applied : {'conditions' : {'cough', 'fever'}, 'conclusion' : 'flu'}

New fact inferred: flu

Final facts : {'cough', 'flu', 'fever'}

Inferred facts: {'flu'}

a) Occupation(Emily, Surgeon) ∨ Occupation(Emily, Lawyer)

b) Occupation(Joe, Actor) ∧ ∃o (o ≠ Actor ∧ occupation(Joe, o))

c) ∀P (Occupation (P, Surgeon) → Occupation (P, Doctor))

d) ¬∃P (Occupation (P, Lawyer) ∧ Customer (Joe, P))

e) ∃P (Occupation (P, Lawyer) ∧ Boss (P, Emily))

```
Class Forward_reasoninig:
        self.rules = rules  # List of rules (condition -> result)
        self.facts = set(facts)  # Known facts

        def infer(self): applied_rules = True

        while applied_rules: applied_rules = False for rule in
        self.rules:
        condition, result = rule
        if condition.issubset(self.facts) and result not in
        self.facts: self.facts.add(result)
        applied_rules = True
        print(f"Applied rule: {condition} -> {result}") return
        self.facts

        # Define rules as (condition, result) where condition
        is a set rules = [
        ({"A"}, "B"),
        ({"B"}, "C"),
        ({"C", "D"}, "E"), ({"E"}, "F")
        ]

        # Define initial facts facts = {"A", "D"}
        # Initialize and run forward reasoning reasoner =
        ForwardReasoning(rules, facts) final_facts =
        reasoner.infer()

        print("\nFinal facts:") print(final_facts)
```

Output:

```
Applied rule: {'A'} -> B
Applied rule: {'B'} -> C
Applied rule: {'C', 'D'} -> E
Applied rule: {'E'} -> F

Final facts:
{'C', 'E', 'B', 'F', 'A', 'D'}
```

Program 9

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

29/11/2024

WEEK - 12

Q: Creating a knowledge Base using propositional logic and proving query using resolution.

Initialize knowledge_base with propositional logic statements

Input Query:

Convert knowledge_base and query into CNF.

Add ¬query to CNF_clauses.

while True:

Select two clauses from CNF_clauses.

Resolve the clauses to produce a new clause

If new_clause is empty:

print("Query is proven using resolution")

break

If new_clause is not already in CNF_clauses:

Add new_clause to CNF_clauses

If no new_clause can be generated:

print("Query can't be proven using resolution")

break

20/12/24

O/p:

For knowledge_base = ["A", "B", "A~B⇒C", "C⇒D"]

Query = "D"

Query is proven using resolution

```
# 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)")
    1
    # 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. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be inferred)
    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 we've reached our goal
if 'Criminal(Robert)' in KB:
print("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)
Robert is a criminal!
```

Program 10

Implement Alpha-Beta Pruning.

29/11/2024

WEEK-09

Q: Implement Alphabeta Pruning

```
function alpha_beta_pruning(node, depth, alpha, beta,
                                        maximizing_player):
        //returns an action.
        v ← MAX_VALUE(state, -∞, +∞)
        return the action in ACTIONS(state) with value v
function MAX_VALUE(state, α, β) returns a utility value
    if TERMINAL_TEST(state) then return UTILITY(state)
    v ← -∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN_VALUE(RESULT(s,a), α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
function MIN_VALUE(state, α, β) returns a utility value
    if TERMINAL_TEST(state) then return UTILITY(state)
    v ← +∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX_VALUE(RESULT(s,a), α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

O/P

For tree = [[3, 5, 6], [9, 1, 2], [0, 7, 4]]

Optimal Value: 6

```
# 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
1
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()))
2
# 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)
if _name___== "_main_":
main()
```

Output:

```
Enter numbers for the game tree (space-separated): 10 9 14 18 5 4 50 3
Final Result of Alpha-Beta Pruning: 50
```