VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Santosh H Jambagi (1BM22CS244)

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 **Santosh H Jambagi (1BM22CS244),** 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.

Lab faculty Incharge Name Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/SantoshJambagi2004/Artificial-Intelligence-Lab

<u>Program 1</u>
Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:

1) Tic-Tac-Toe

a) Algorithm: Tic-Tac_Toe()

step 1: Create a 3x3 3Dimensional array filled with underscores ('_') to indicate empty spaces.

step 2: Set current player to 'X'

Validate the input: if row<0 or row>2 or board[i][j]=='_'

b) check for winner

check all rows, columns and diagonals to see if any player having any three symbols in line.

c) check for Draw

If no winner is found, check if the board is full.

d) Switch players

If the game is still ongoing switch current player from 'X' to 'O' or from 'O' to 'X'.

e) Update the board

place the current player's symbol ('X' or 'O') in the selected cell.

f) Display the board

print the board along with its current state whether winner is found or not. Or its a draw.

1/10/24

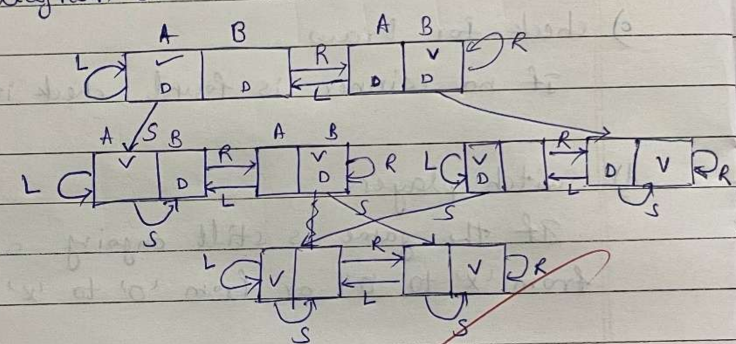
LAB - 1

→ Implement Vacuum World cleaner

function REFLEX_VACUUM_AGENT ([location, status]) returns an ~~funct~~ action.

If status = Dirty then return Suck
else If location = A then return Right
else If location = B then return Left

state-space diagram of vacuum world:



Algorithm:

Initial Environment:
- Define grid with cells that are either dirty(1) or clean(0).
- set the initial position of the vacuum cleaner.
- set the goal state: {'A': 0, 'B': 0}.

The vacuum cleaner can perform the following actions:

1) clean: clean the current cell.
2) Move left: Move to the left cell if it exists
3) Move right: Move to the right cell if it exists.

Initial state : [1 , 1]

- If the initial state matches the goal state then exit the function.
- Increment the cost ~~for the~~ one suck operation.
- goal state = {'A' : 0 , 'B' : 0}

1/10/24

Code: 1: Tic - Tac - Toe

```
import numpy as np

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

current_player='X'
flag=0

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

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

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

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

    if n==9:
```

```
    print("Draw")
```

tic_tac_toe()

Output:

Win :

```
_ | _ | _
_ | _ | _
_ | _ | _
Player X, enter the row (0, 1, 2): 0
Player X, enter the column (0, 1, 2): 0
X | _ | _
_ | _ | _
_ | _ | _
Player O, enter the row (0, 1, 2): 1
Player O, enter the column (0, 1, 2): 0
X | _ | _
O | _ | _
_ | _ | _
Player X, enter the row (0, 1, 2): 0
Player X, enter the column (0, 1, 2): 1
X | X | _
O | _ | _
_ | _ | _
Player O, enter the row (0, 1, 2): 1
Player O, enter the column (0, 1, 2): 1
X | X | _
O | O | _
_ | _ | _
Player X, enter the row (0, 1, 2): 0
Player X, enter the column (0, 1, 2): 2
X | X | X
O | O | _
_ | _ | _
Player X wins!
```

Draw :

```
Player O, enter the row (0, 1, 2): 0
Player O, enter the column (0, 1, 2): 2
O | X | O
X | _ | _
_ | _ | _
Player X, enter the row (0, 1, 2): 2
Player X, enter the column (0, 1, 2): 0
O | X | O
X | _ | _
X | _ | _
Player O, enter the row (0, 1, 2): 1
Player O, enter the column (0, 1, 2): 1
O | X | O
X | O | _
X | _ | _
Player X, enter the row (0, 1, 2): 1
Player X, enter the column (0, 1, 2): 2
O | X | O
X | O | X
X | _ | _
Player O, enter the row (0, 1, 2): 2
Player O, enter the column (0, 1, 2): 1
O | X | O
X | O | X
X | O | _
Player X, enter the row (0, 1, 2): 2
Player X, enter the column (0, 1, 2): 2
O | X | O
X | O | X
X | O | X
It's a draw!
```

2. Vacuum Cleaner :

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

  if state[location]==1:
    state[location]=0
    cost+=1
```

```
    state[location]=(int(input('Is room '+ str(location) +' still dirty : ')))

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

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

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

Output :

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

Program 2

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

Algorithm:

LAB - 2

8 puzzle problem using BFS & DFS

Initial Setup:

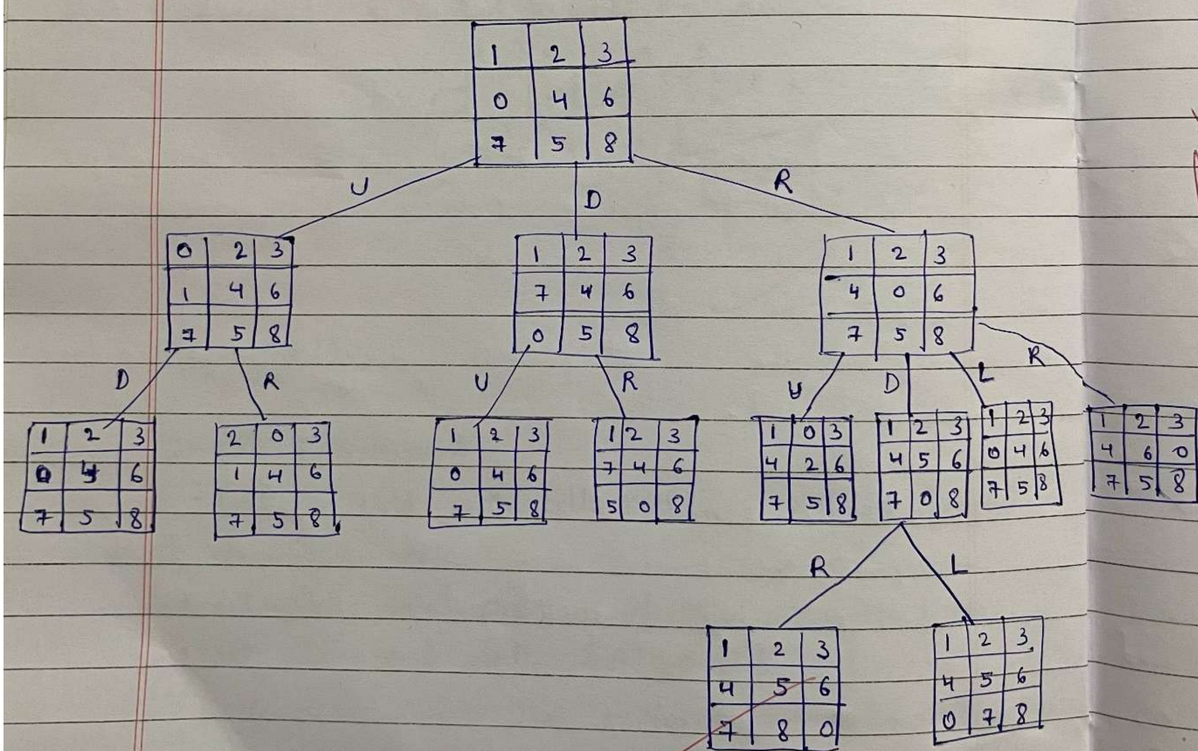
→ Create a 3x3 grid.

→ Represent each state with a tuple containing 9 numbers between (0 to 8)

Goal State:

→ Set a Goal state =

1	2	3
4	5	6
7	8	0



Date: Page:

Finding empty tile:

Find 0 - i.e Blank tile.

Generating possible moves

For each possible direction (up, down, left, right)

- check if the moves is valid

If valid, create a new state by swapping the blank and record the move.

BFS Loop:

While the queue is not empty:

- Dequeue the current state and its path.
- check if the current state matches goal state

10-24

2	3
6	0
5	8

22/10/24

Lab program: Implement Iterative Deepening search

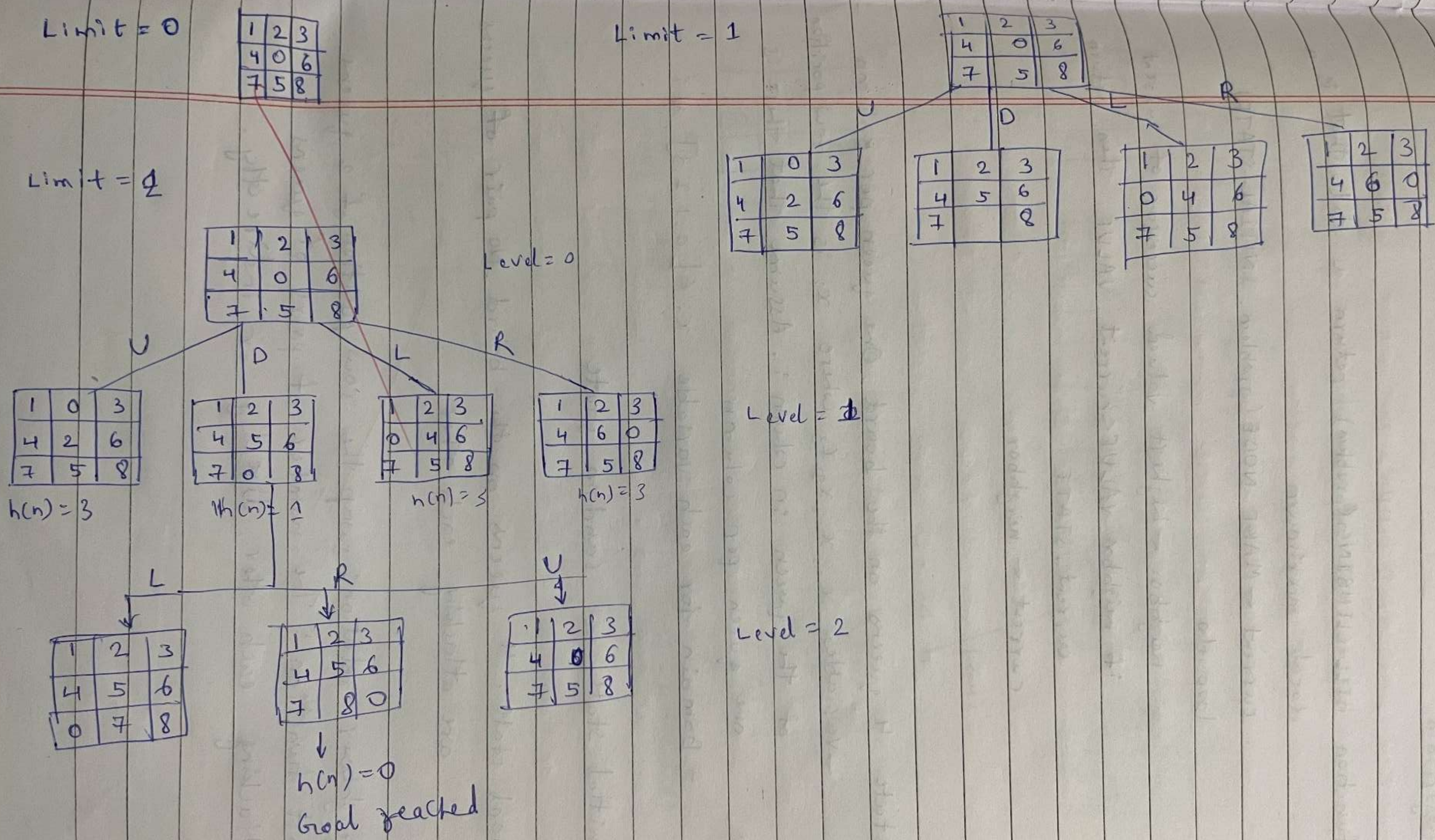
Algorithm

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

function ITERATIVE-DEEPENING-SEARCH(problem)
return a solution or failure
for depth = 0 to ∞ do
result $\leftarrow$ DEPTH-LIMITED-SEARCH(problem, depth)
if result $\neq$ cut-off then return result.

Goal state:

1	2	3
4	5	6
7	8	0



Limit = 0
Limit = 1
Limit = 2
Level = 0
Level = 1
Level = 2
U
D
L
R
h(n) = 3
h(n) = 1
h(n) = 5
h(n) = 3
h(n) = 0
Goal reached

Code:
1: DFS

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

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

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

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

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


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

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

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

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

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


 vis[row][col] = 0
 return False


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

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

Output :

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1 2 3
4 5 6
7 8 0

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

2 : Iterative deepening search

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

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

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

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

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

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

```
    if depth <= 0:
        return None

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

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

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

    initial_state = PuzzleState(initial_board, empty_tile_pos)

    solution = iddfs(initial_state, goal_board, max_depth)

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

Output :

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Goal not found within max depth.
```

Program 3

Implement A* search algorithm

Algorithm:

Q) For 8 puzzle problem using A* implementation to calculate $f(n)$ using

a) $g(n)$ = depth of a node
$h(n)$ = heuristic value → no of misplaced tiles

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

b) $g(n)$ = depth
$h(n)$ = heuristic value
↓
Manhattan distance

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

Algorithm for A* search:

step 1: place the starting node in the OPEN list.

step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stop.

step 3: Select the node from the open list which has the smallest value of evaluation function $(g+h)$, if node n is goal node then return success and stop. otherwise

step 4: Expand node n and generate all its successors and put n into the closed list. For each successor 'n', check whether 'n' is already in the OPEN or CLOSED list, if not then compute evaluation function for 'n' and place into OPEN list.

Bafna Gold
Date: Page:

step 5: Else if node 'n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n) value.

step 6: Return to step 2.

15/10/24

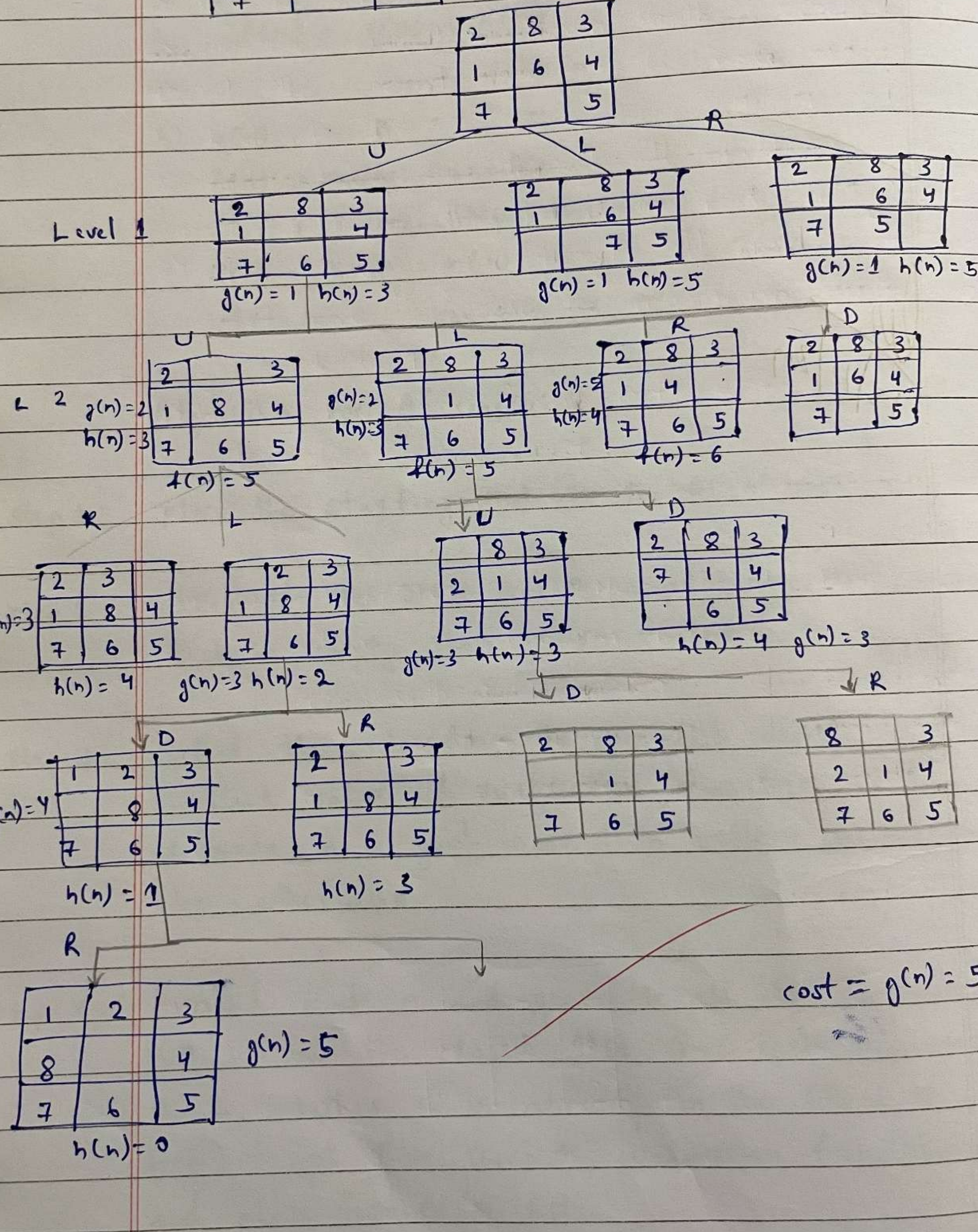
Q) Draw the state space diagram for misplaced tiles

Initial state

2	8	3
1	6	4
7		5

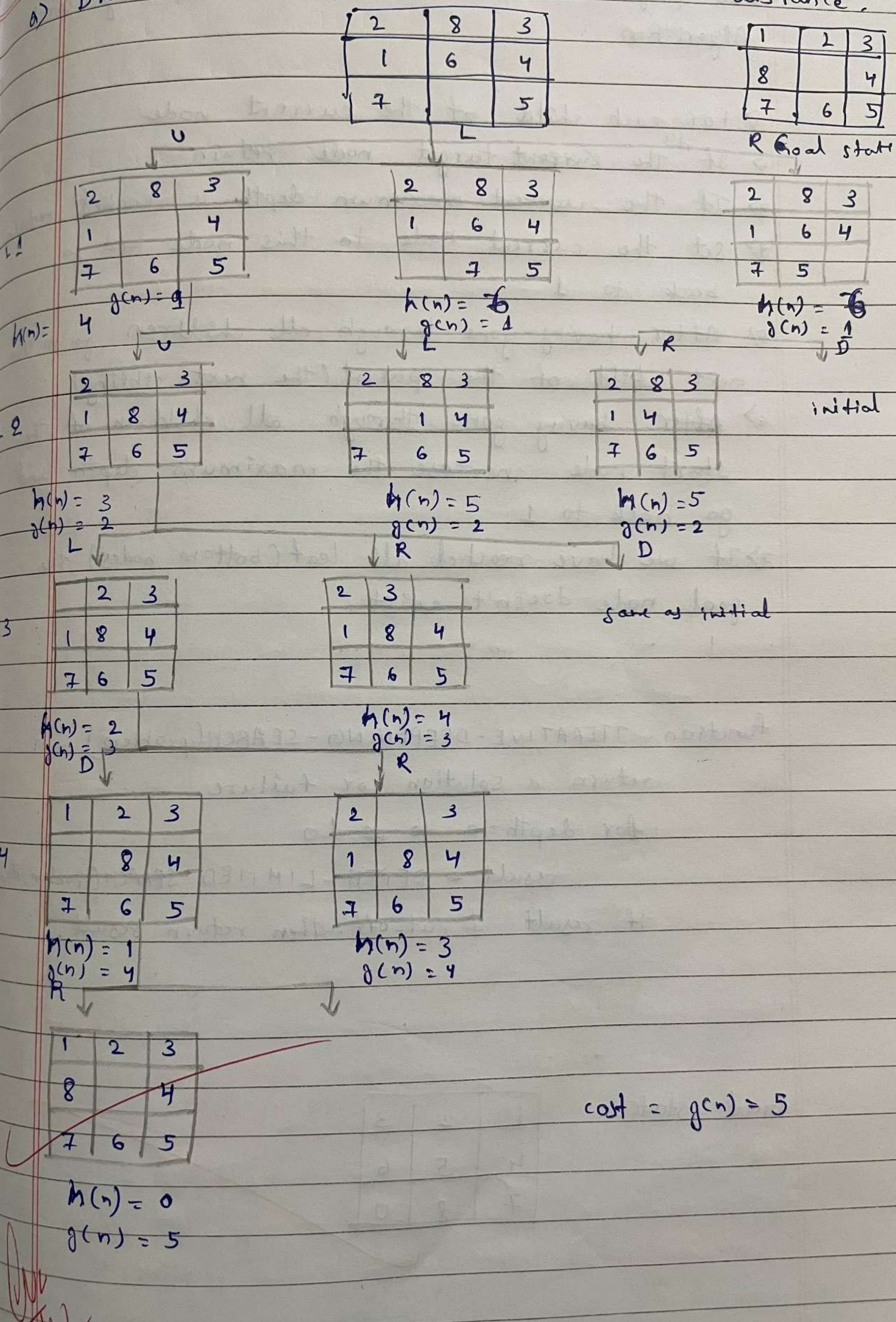
Goal state

1	2	3
8		4
7	6	5



cost = g(n) = 5

Q) Draw the state space tree for Manhattan distance.



cost = g(n) = 5

Code:

Misplaced Tiles :

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

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

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

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

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

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

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

```
    while open_list:

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


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

        if current_node.heuristic() == 0:

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

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

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

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

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

    return None

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

if solution_path:
    print("Solution path:")
    for step in solution_path:
```

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

Output :

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

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

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

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

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

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

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

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

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

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

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

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

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

Code :

Manhattan distance approach

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

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

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

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

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

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

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

    while open_list:

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


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

        if current_node.heuristic() == 0:

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

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

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

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

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

    return None
```

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

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

Output :

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

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

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

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

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

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

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

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

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

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

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

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

22/10/24

Implement Hill climbing search to solve N-Queens problem.

Algorithm:

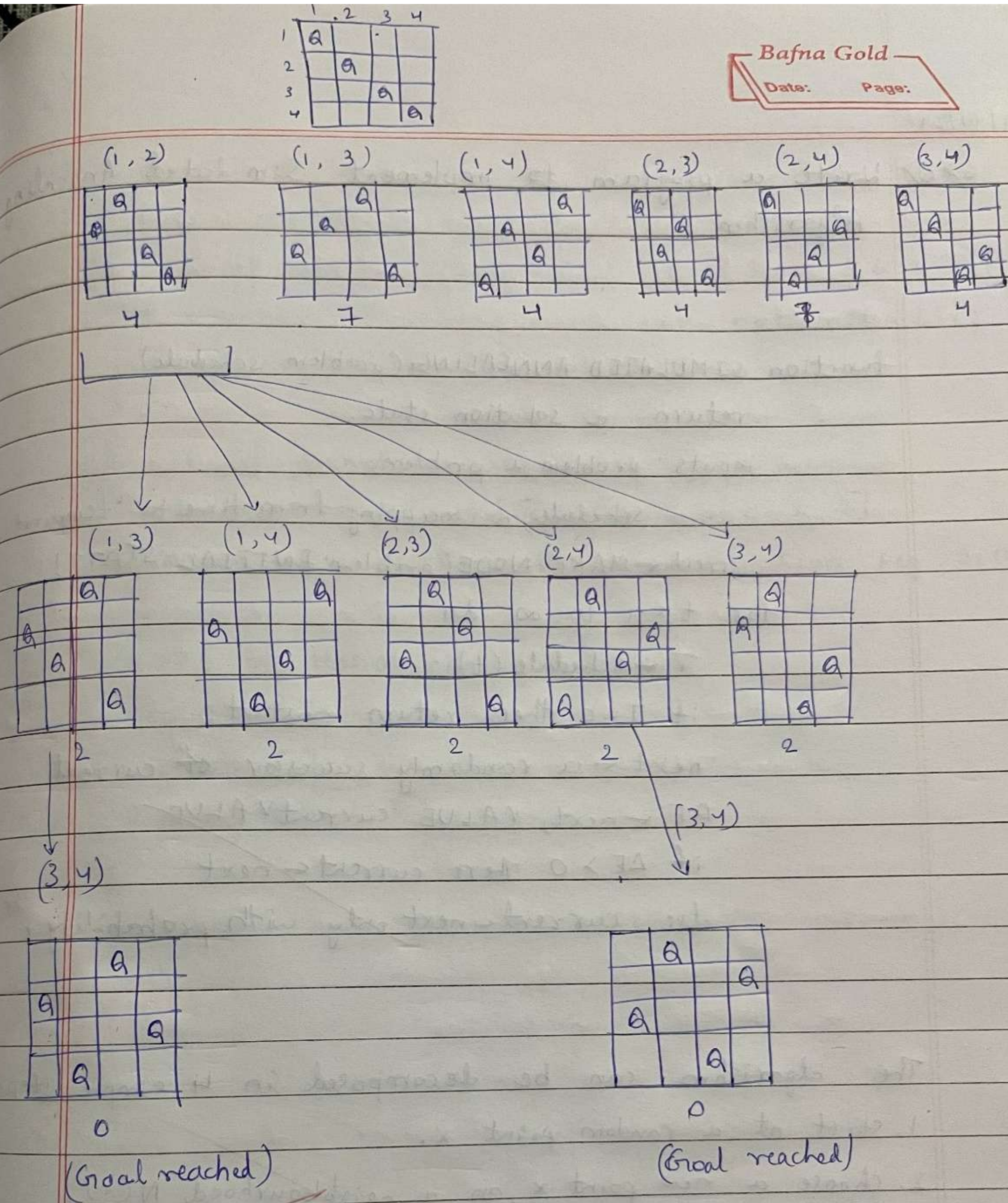
function HILL-CLIMBING(problem) return a state that is a local maximum.
current ← MAKE-NODE(problem. INITIAL-STATE)
loop do
neighbor ← highest valued successor of current
if neighbor. VALUE ≤ current. VALUE then return current. STATE
current ← neighbor.

- state: 4 queens on the board. One queen per column
  - variables: x_0, x_1, x_2, x_3 where x_i is the row position of the queen in column i. Assume that there is one queen per column
  - Domain for each variable $x_i \in [0,1,2,3], \forall i$

- Initial state: a random state

- Goal state: 4 queens on the board. No pair of queens are attacking each other

- Neighbour relation: swap the row position of 2 queens
- Cost function: The number of pairs of queens attacking each other, directly or indirectly.



1 2 3 4
(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)
4
7
4
4
4
(1, 3)
(1, 4)
(2,3)
(2,4)
(3, 4)
2
2
2
2
2
(3, 4)
(3, 4)
0
(Goal reached)
0
(Goal reached)

Code:

```
import random
def calculate_cost(board):

    n = len(board)

    attacks = 0

    for i in range(n):

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

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

                attacks += 1

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

                attacks += 1

    return attacks


def get_neighbors(board):

    neighbors = []

    n = len(board)

    for col in range(n):

        for row in range(n):

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

                new_board = board[:]

                new_board[col] = row

                neighbors.append(new_board)

    return neighbors
```

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

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

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

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

            board = best_neighbor
```

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

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

    return board, current_cost

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

```
        print(' '.join(display_board[row]))  # Print the board
    print(f"Cost: {cost}\n")


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

Output :

```
Enter the number of queens (N): 4
Enter the initial state (row numbers for each column, space-separated): 0 1 2 3
Initial board configuration:
Q . . .
. Q . .
. . Q .
. . . Q
Cost: 6

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

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

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

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

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

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

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

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

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

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

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

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

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

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

29/10/24

6> Q) Write a program to implement Simulated Annealing Algorithm.

function SIMULATED_ANNEALING(problem, schedule)
    return a solution state.
    inputs: problem, a problem
        schedule, a mapping from time to "temparature"
    current ← MAKE-NODE(problem. INITIAL-STATE)
    for t ← 1 to ∞ do
        T ← schedule (t)
        if T = 0 then return constant
        next ← a randomly successor of current
        ΔE ← next, VALUE . current VALUE
        if ΔE > 0 then current ← next
        else current ← next only with probability $e^{\Delta E/T}$

The algorithm can be decomposed in 4 simple steps:

1. start at a random point x.
2. choose a new point x, on a neighbourhood N(x).
3. Decide whether or not to move to the new point x, The decision will be made based on the probability function $P(x, x_j, T)$

$$P(x, x_j, T) = \begin{cases} 1 & si \quad F(x_j) \geq F(x) \\ e^{\frac{F(x_j) - F(x)}{T}} & si \quad F(x_j) < F(x) \end{cases}$$

4. Reduce T.

Bafna Gold
Date: Page:

Output:

8 Queens Problem:

The best position found is: [4 1 7 0 3 6 2 5]

The number of queens that are not attacking each other is : 8.0

Tower of Hanoi:

Best state (final configuration): [2 2 2]

Number of correct disks on destination peg: 3.0

Tower of Hanoi configuration:

peg 0: [ ]

peg 1: [ ]

peg 2: [0, 1, 2]

24/10/24

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

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

objective = mlrose.CustomFitness(queens_max)

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

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

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

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

Output :

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

Program 6

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

Algorithm:

12/11/24

LAB 6

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

```
function TT-ENTAILS?(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in proporsitional
          logic
          α, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in KB and α.
  return TT-CHECK-ALL(KB, α, symbols, { })


function TT-CHECK-ALL(KB, α, symbols, model) returns
         true or false
  if EMPTY? (symbols) then
      if PL-TRUE? (KB, model) then return PL-TRUE?(α, model)
      else return true || when KB is false, always return true

  else do
      P ← FIRST(symbols)
      rest ← REST(symbols)
      return(TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
             and
             TT-CHECK-ALL(KB, α, rest, model ∪ {P = false}))
```

Bajna Gold

Date: Page:

Propositional Interface:

Ex: $\alpha = A \vee B$ $KB = (A \vee C) \wedge (B \vee \neg C)$

checking that $KB \models \alpha$

A	B	C	$A \vee C$	$B \vee \neg C$	KB	α
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	F	T
$\boxed{F}$	$\boxed{T}$	$\boxed{T}$	T	T	$\boxed{T}$	$\boxed{T}$
$\boxed{T}$	$\boxed{F}$	$\boxed{F}$	T	T	$\boxed{T}$	$\boxed{T}$
T	F	T	T	F	F	T
$\boxed{T}$	$\boxed{T}$	$\boxed{F}$	T	T	$\boxed{T}$	$\boxed{T}$
$\boxed{T}$	$\boxed{T}$	$\boxed{T}$	T	T	$\boxed{T}$	$\boxed{T}$

O/p seen

12/11/24

Code:

```
import pandas as pd

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

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

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

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

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

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

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

# Display the styled table
styled_table
```

Output :

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

Program 7

Implement unification in first order logic

Algorithm:

LAB 7

Q) Implement Unification in First Order Logic.

Algorithm: Unify(Ψ_1, Ψ_2)

step 1: If Ψ_1 or Ψ_2 is a variable or constant, then:

a) If Ψ_1 or Ψ_2 are identical, then return NIL.

b) Else if Ψ_1 is a variable,

a. then if Ψ_1 occurs in Ψ_2, then return FAILURE

b. Else return $\{(\Psi_2/\Psi_1)\}$.

c) Else if Ψ_2 is a variable,

a. If Ψ_2 occurs in Ψ_1 then return FAILURE

b. Else return $\{(\Psi_1/\Psi_2)\}$.

d) Else return FAILURE.

step 2: If the initial predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

step 3: If Ψ_1 and Ψ_2 have a different number of arguements then return FAILURE.

step 4: Set Substitution set(SUBST) to NIL.

step 5: For i=1 to the number of elements in Ψ_1.

a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

b) If S = failure then returns failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 & L2

b. SUBST = APPEND(S, SUBST)

Step 6: Return SUBST

Code:

```
import re

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


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


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

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

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

```
            return "FAILURE"

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


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


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


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

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

```
            return [predicate] + arguments
        return term

    return parse_term(input_str)


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

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

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

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

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


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

Output :

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

Program 8

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

Algorithm:

Date: 26/11/24

LAB 8

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

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 $\leftarrow$ { }

for each rule in KB do:

($p_1 \wedge \ldots \wedge p_n \Rightarrow q$) $\leftarrow$ STANDARDIZE-VARIABLES(rule)

for each θ such that SUBST($\theta, p_1 \wedge \cdots \wedge p_n$) = SUBST ($\theta, p_1' \wedge \ldots \wedge p_n'$)

for some($p_1' \cdot \ldots p_n'$ in KB

$q' \leftarrow$ SUBST(θ, q)

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

add q' to new

$\phi \leftarrow$ UNIFY(q', α)

if ϕ is not fail then return ϕ

add new to KB

return false

Representation in FOL:

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

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

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

→ Country A has some missiles

∃x Owns(A, x) ∧ Missile(x)

Existential Instantiation, introducing a new constant T1:

Owns(A, T1)

Missile(T1)

→ All the missiles were sold to country A by Robert

∀x Missile(x) ∧ Owns(A, x) ⇒ Sells(Robert, x, A)

→ Missiles are weapons · Robert is an American

Missiles(x) ⇒ weapons(x) American(Robert)

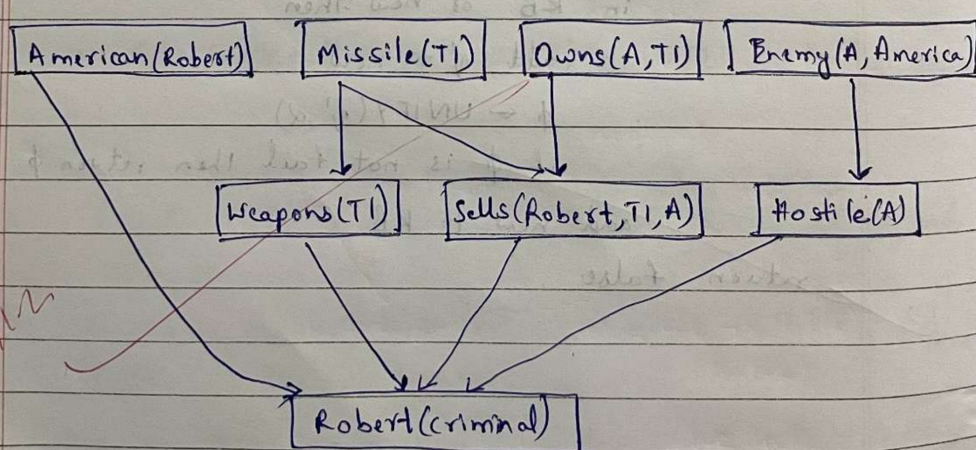
→ Enemy of America is known as hostile

∀x Enemy(x, America) ⇒ Hostile(x)

→ The country A, an enemy of America

Enemy(A, America)

To Prove: Robert is Criminal Criminal(Robert)



American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r)

⇒ Criminal(p)

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

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

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

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

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

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

# Initialize the knowledge base
kb = KnowledgeBase()

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

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

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

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

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

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

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

Output :

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

Program 9

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

Algorithm:

LAB 9

Q) Convert a given first order logic statement into Resolution

Basic steps for proving a conclusion S given premises Premise$_1$, ... Premise$_n$

1. Convert all sentences to CNF
2. Negate conclusion S & convert result to CNF
3. Add negated conclusion S to the premise clauses
4. Repeat until contradiction or no progress is made
   a. Select 2 clauses (call them parent clauses)
   b. Resolve them together, performing all required unifications
   c. If resolvent is the empty clause, a contradiction has been found (i.e S follows from the premises)
   d. If not, add resolvent to the premises

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

Representation in FOL:

Given KB or premises:

a. John likes all kind of food	a. $\forall x: food(x) \rightarrow likes(John, x)$
b. Apple and vegetables are food	b. $food(Apple) \wedge food(vegetables)$
c. Anything anyone eats and not killed is food	c. $\forall x \forall y: eats(x,y) \wedge \neg killed(x) \rightarrow food(y)$
d. Anil eats peanuts & still alive	d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. Harry eats everything that Anil eats	e. $\forall x: eats(Anil, x) \rightarrow eats(Harry, x)$
f. Anyone who is alive implies not killed	f. $\forall x:$ ~~alive(x) → ¬ killed(x)~~ $\neg killed(x) \rightarrow alive(x)$
g. Anyone who is not killed implies alive	g. $\forall x: alive(x) \rightarrow \neg killed(x)$
h. Prove by resolution that: John likes peanuts	h. $likes(John, peanuts)$

① Eliminate Implication:

a. $\forall x \neg food(x) \vee likes(John, x)$

b. $food(Apple) \wedge food(vegetables)$

c. $\forall x \forall y \neg[eats(x,y) \wedge \neg killed(x)] \vee food(y)$

d. $eats(Anil, Peanuts) \wedge alive(Anil)$

e. $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$

f. $\forall x \neg[\neg killed(x)] \vee alive(x)$

g. $\forall x \neg alive(x) \vee \neg killed(x)$

h. $likes(John, Peanuts)$

② Move Negation (¬) inwards & write

c. $\forall x \forall y \neg eats(x,y) \vee killed(x) \vee food(y)$

f. $\forall x [killed(x)] \vee alive(x)$

③ Rename variables or standardize variables:

a. $\forall x \neg food(x) \vee likes(John, x)$

b. $food(Apple) \wedge food(vegetables)$

c. $\forall y \forall z \neg eats(y,z) \vee killed(y) \vee food(z)$

d. $eats(Anil, Peanuts) \wedge alive(Anil)$

e. $\forall w \neg eats(Anil, w) \vee eats(Harry, w)$

f. $\forall g [killed(g)] \vee alive(g)$

g. $\forall k \neg alive(k) \vee \neg killed(k)$

h. $likes(John, Peanuts)$

④ Drop Universal Quantifier

a. $\neg food(x) \vee likes(John, x)$

b. $food(Apple)$

c. $food(vegetables)$

d. $\neg eats(y,z) \vee killed(y) \vee food(z)$

e. $eats(Anil, Peanuts)$

f. $alive(Anil)$

g. $\neg eats(Anil, w) \vee eats(Harry, w)$

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

i. $\neg alive(k) \vee \neg killed(k)$

j. $likes(John, Peanuts)$

Proof by Resolution:

$\neg likes(John, Peanuts)$ $\neg food(x) \vee likes(John, x)$

$\{Peanuts/x\}$

$\neg food(Peanuts)$ $\neg eats(y,z) \vee killed(y) \vee food(z)$

$\{Peanuts/z\}$

$\neg eats(y, Peanuts) \vee killed(y)$ $eats(Anil, Peanuts)$

$\{Anil/y\}$

$killed(Anil)$ $\neg alive(k) \vee \neg killed(k)$

$\{Anil/k\}$

$\neg alive(Anil)$ $alive(Anil)$

Code:
```
KB = {
    "food(Apple)": True,
    "food(vegetables)": True,
    "eats(Anil, Peanuts)": True,
    "alive(Anil)": True,
    "likes(John, X)": "food(X)",  # Rule: John likes all food
    "food(X)": "eats(Y, X) and not killed(Y)",  # Rule: Anything eaten and not killed is food
    "eats(Harry, X)": "eats(Anil, X)",  # Rule: Harry eats what Anil eats
    "alive(X)": "not killed(X)",  # Rule: Alive implies not killed
    "not killed(X)": "alive(X)",  # Rule: Not killed implies alive
}

# Function to evaluate if a predicate is true based on the KB
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]

    # If it's a derived rule
    if predicate in KB:
        rule = KB[predicate]
        if " and " in rule:  # Handle conjunction
            sub_preds = rule.split(" and ")
            return all(resolve(sub.strip()) for sub in sub_preds)
        elif " or " in rule:  # Handle disjunction
            sub_preds = rule.split(" or ")
            return any(resolve(sub.strip()) for sub in sub_preds)
        elif "not " in rule:  # Handle negation
            sub_pred = rule[4:]  # Remove "not "
            return not resolve(sub_pred.strip())
        else:  # Handle single predicate
            return resolve(rule.strip())

    # If the predicate is a specific query (e.g., likes(John, Peanuts))
    if "(" in predicate:
        func, args = predicate.split("(")
        args = args.strip(")").split(", ")
        if func == "food" and args[0] == "Peanuts":
            return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)")
        if func == "likes" and args[0] == "John" and args[1] == "Peanuts":
            return resolve("food(Peanuts)")

    # Default to False if no rule or fact applies
    return False

# Query to prove: John likes Peanuts
```

```
query = "likes(John, Peanuts)"
result = resolve(query)

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

Output:

```
Does John like peanuts? Yes
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:

Date: 26/11/24

LAB 10

Q) Implement Alpha Beta Pruning Algorithm

- Alpha(α) – Beta(β) proposes to find the optimal path without looking at every node in the game tree.

- Max contains Alpha (α) and Min contains Beta(β) bound during the calculation

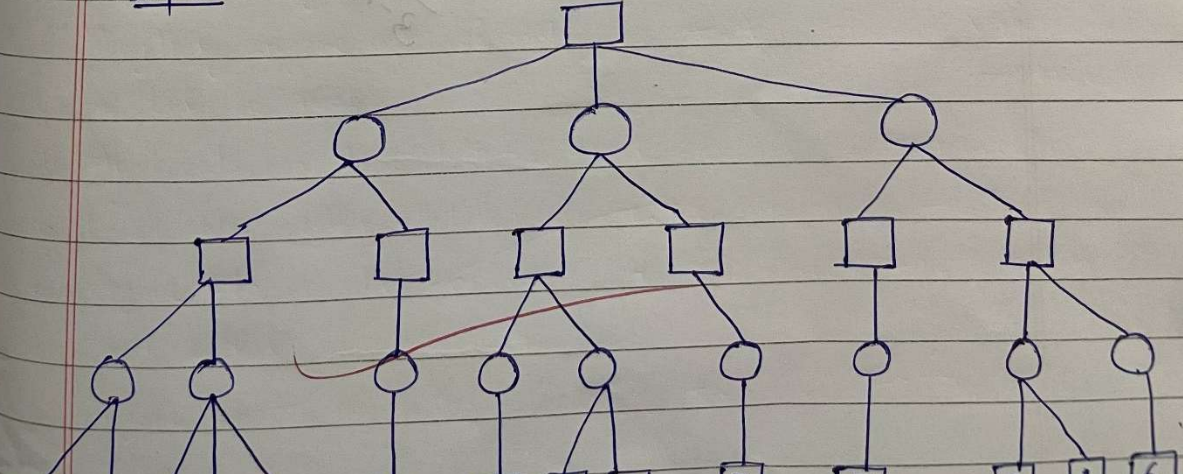
- In both MIN and MAX node, we return when $\alpha \geq \beta$ which compares with its parent node only.

- Both minimax and Alpha(α) - Beta(β) cut-off gives same path

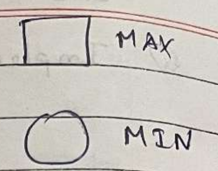
- Alpha(α) - Beta(β) gives optimal solution as it takes less time to get the value for the root node.

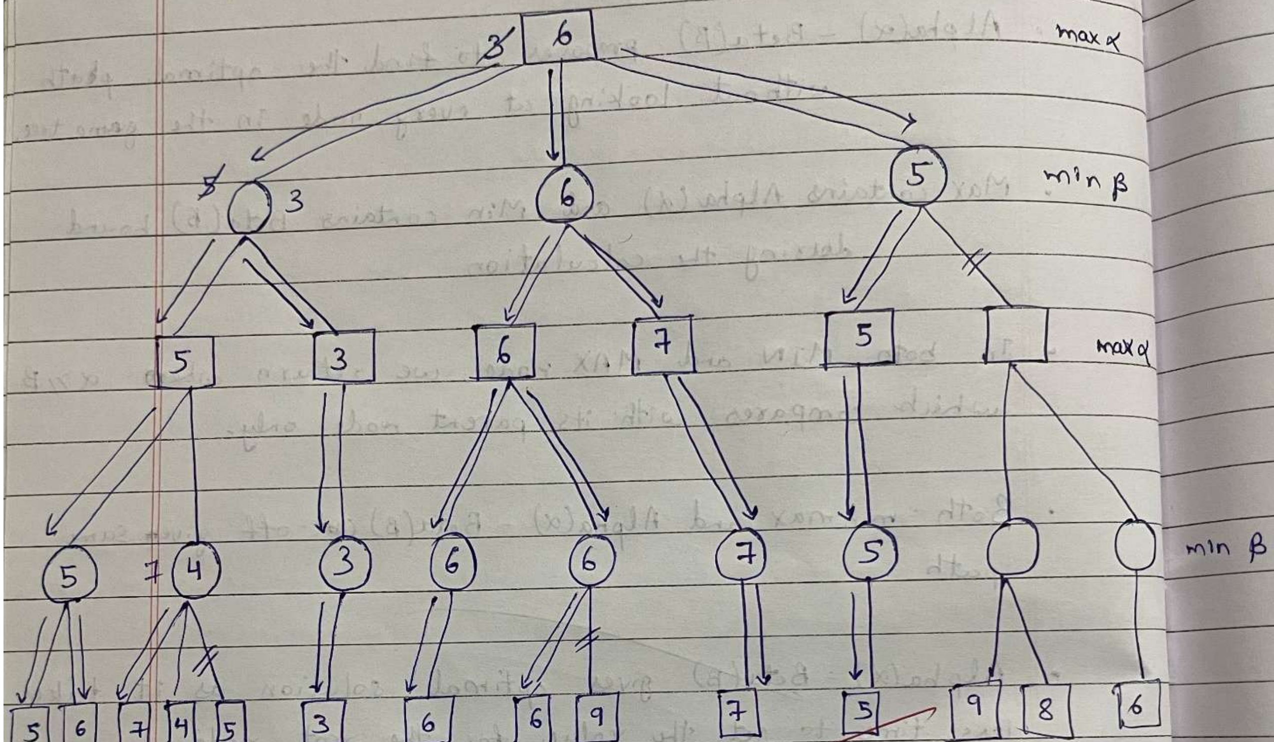
MAX MIN

Input:



Output:





3-12-24

Code:

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

    Parameters:
    node (dict): The current node in the decision tree, with the following structure:
        {
            'value': int,
            'left': dict or None,
            'right': dict or None
        }
    depth (int): The current depth in the decision tree.
    is_maximizing (bool): Flag to indicate whether the current player is the maximizing player.

    Returns:
    int: The utility value of the current node.
    """
    # Base case: Leaf node
    if node['left'] is None and node['right'] is None:
        return node['value']

    # Recursive case
    if is_maximizing:
        best_value = -math.inf
        if node['left']:
            best_value = max(best_value, minimax(node['left'], depth + 1, False))
        if node['right']:
            best_value = max(best_value, minimax(node['right'], depth + 1, False))
        return best_value
    else:
        best_value = math.inf
        if node['left']:
            best_value = min(best_value, minimax(node['left'], depth + 1, True))
        if node['right']:
            best_value = min(best_value, minimax(node['right'], depth + 1, True))
        return best_value

# Example usage
decision_tree = {
    'value': 5,
    'left': {
        'value': 6,
        'left': {
            'value': 7,
            'left': {
```

```
            'value': 4,
            'left': None,
            'right': None
        },
        'right': {
            'value': 5,
            'left': None,
            'right': None
        }
    },
    'right': {
        'value': 3,
        'left': {
            'value': 6,
            'left': None,
            'right': None
        },
        'right': {
            'value': 9,
            'left': None,
            'right': None
        }
    }
},
'right': {
    'value': 8,
    'left': {
        'value': 7,
        'left': {
            'value': 6,
            'left': None,
            'right': None
        },
        'right': {
            'value': 9,
            'left': None,
            'right': None
        }
    },
    'right': {
        'value': 8,
        'left': {
            'value': 6,
            'left': None,
            'right': None
        },
        'right': None
```

```
        }
    }
}

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

Output:

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