VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Venkatesh Vinay Chandle(1BM22CS325)

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 **Venkatesh Vinay Chandle(1BM22CS325),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence(23CS5PCAIN) work prescribed for the said degree.

Prof. Rashmi H Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Github Link: https://github.com/venkateshchandle/AI

Table of contents

Program 1 - Tic Tac toe

Algorithm

Lab - 01 Date: 4/10/2024

1 Tic-Tac-Toe implementation using python

Pseudocode

```
function minimax(node, depth, isMaximizingPlayer)
    if node is a terminal state
        return evaluate(node)
    if isMaximizingPlayer:
        bestValue = -inf
        for each child in node:
            value = minimax(child, depth, false)
            bestvalue = max(bestvalue, value)
        return bestvalue
    else:
        bestvalue = +inf
        for each child in node
            value = minimax(child, depth, true)
            bestvalue = min(bestvalue, value)
        return bestvalue
```

Code

```
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])2
    print('-+-+-')
    print(board[7] + '|' + board[8] + '|' + board[9])
    print('\n')

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

def checkWin():
    win_conditions = [
        (1, 2, 3), (4, 5, 6), (7, 8, 9),  # Rows
        (1, 4, 7), (2, 5, 8), (3, 6, 9),  # Columns
        (1, 5, 9), (3, 5, 7)  # Diagonals
    ]
    for a, b, c in win_conditions:
        if board[a] == board[b] == board[c] and board[a] != ' ':
            return True
    return False
```

```python
def checkMoveForWin(move):
    win_conditions = [
        (1, 2, 3), (4, 5, 6), (7, 8, 9),
        (1, 4, 7), (2, 5, 8), (3, 6, 9),
        (1, 5, 9), (3, 5, 7)
    ]
    for a, b, c in win_conditions:
        if board[a] == board[b] == move and board[a] != ' ':
            return True
    return False


def checkDraw():
    return all(board[key] != ' ' for key in board.keys())


def insertLetter(letter, position):
    if spaceFree(position):
        board[position] = letter
        printBoard(board)
        if checkDraw():
            print('Draw!')
        elif checkWin():
            if letter == 'X':
                print('Bot wins!')
            else:
                print('You win!')
            return
    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: '))
    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
    insertLetter(bot, bestMove)


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

print("Venkatesh Vinay
Chandle")
print("1BM22CS325\n")

while not checkWin() and not checkDraw():
```

```
compMove()
if checkWin() or checkDraw():
    break
```

Program 2 - 8 Puzzle Using BFS

<u>Algorithm</u>

3. Implement puzzle pro...

Algorithm:

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

then return path from initial state to node

else generate all successors of Node and add generated nodes to the back of fringe.

End Loop

Consider initial state and final state

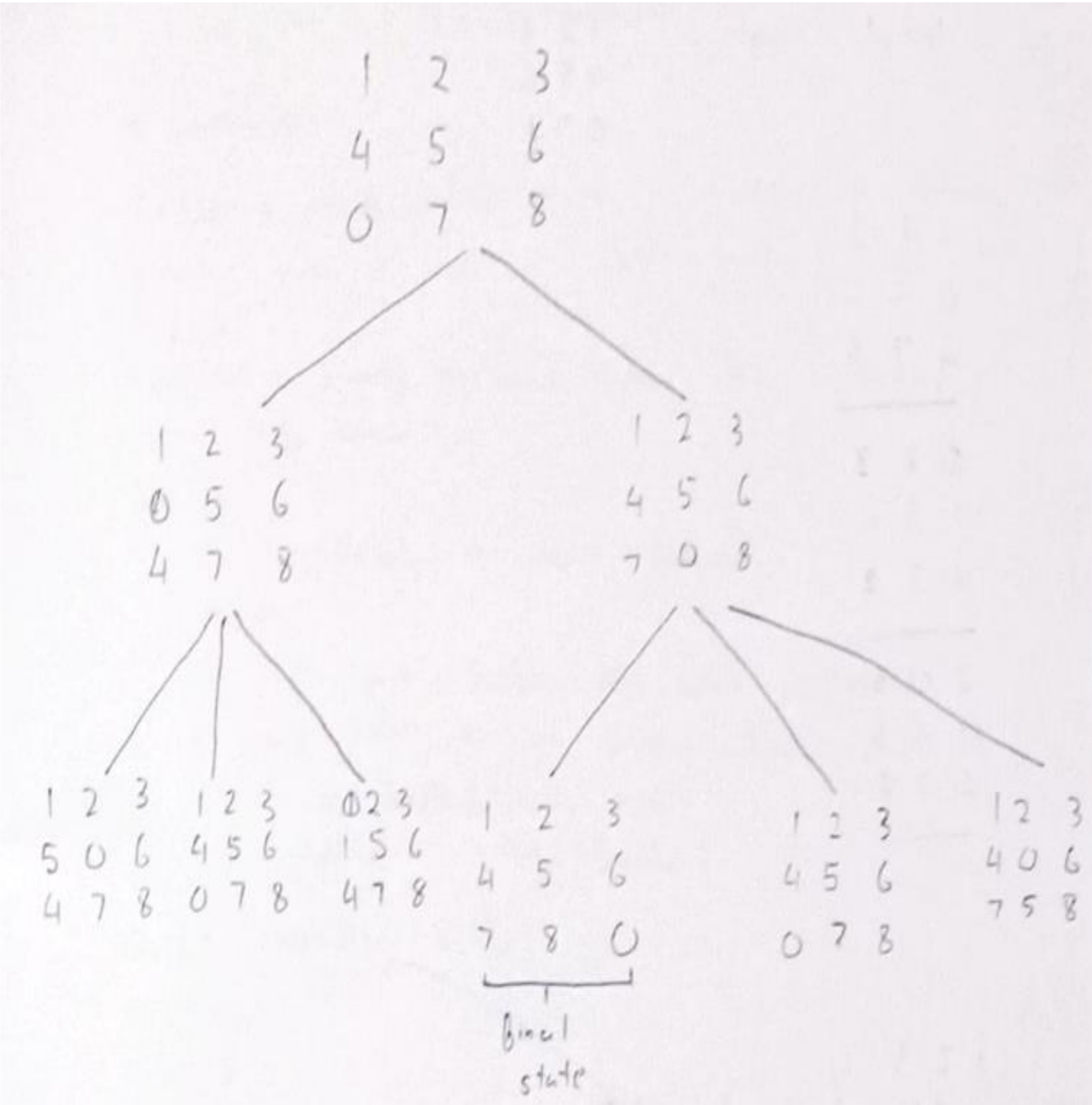
Initial			Goal		
1	2	3	1	2	3
4	5	6	4	5	6
0	7	8	7	8	0

Code

```
#BFS
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_position, path=[]):
        self.board = board
        self.zero_position = zero_position
        self.path = path

    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_possible_moves(self):
        moves = []
        row, col = self.zero_position
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                # Swap zero with the adjacent tile
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] = new_board[new_row * 3 + new_col], new_board[row * 3 + col]
                moves.append(PuzzleState(new_board, (new_row, new_col), self.path + [new_board]))
        return moves

def bfs(initial_state):
    queue = deque([initial_state])
    visited = set()

    while queue:
        current_state = queue.popleft()
        # Show the current board
```

```
        print("Current Board State:")
        print_board(current_state.board)
        print()

        if current_state.is_goal():
            return current_state.path
        visited.add(tuple(current_state.board))

        for next_state in current_state.get_possible_moves():
            if tuple(next_state.board) not in visited:
                queue.append(next_state)

    return None

def print_board(board):
    for i in range(3):
        print(board[i * 3:i * 3 + 3])

def main():
    print("Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'): ")
    user_input = input()
    initial_board = list(map(int, user_input.split()))

    if len(initial_board) != 9 or set(initial_board) != set(range(9)):
        print("Invalid input! Please enter 9 numbers from 0 to 8.")
        return

    zero_position = initial_board.index(0)
    initial_state = PuzzleState(initial_board, (zero_position // 3, zero_position % 3))
    solution_path = bfs(initial_state)

    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found in", len(solution_path), "steps.")
        for step in solution_path:
            print_board(step)
            print()

if __name__ == "__main__":
    main()

print("------------ ")
```

```
print("Venkatesh Vinay
Chandle")
print("1BM22CS325")
```



8 puzzle using DFS

11) Implement 8 puzzle problem using DFS Algorithm

Algorithm :

Let fringe be the list containing the initial state

Loop If fringe is empty, return failure

Node ← remove. first (fringe)

If Node is a goal

then return path from initial state to ~~final state~~ node

else generate all successors of Node & E

add generated Nodes to the front of the fringe

Code

```
from collections import deque

print("Venkatesh Vinay Chandle")
print("1BM22CS325")
print(" ------- ")

def get_user_input(prompt):
    board = []
    print(prompt)
    for i in range(3):
        row = list(map(int, input(f"Enter row {i + 1} (space-separated numbers, use 0 for empty space): ").split()))
        board.append(row)
    return board

def is_solvable(board):
    flattened_board = [tile for row in board for tile in row if tile != 0]
    inversions = 0
    for i in range(len(flattened_board)):
        for j in range(i + 1, len(flattened_board)):
            if flattened_board[i] > flattened_board[j]:
                inversions += 1
    return inversions % 2 == 0

class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
```

```
        self.board = board
        self.empty_tile = self.find_empty_tile()
        self.moves = moves
        self.previous = previous

    def find_empty_tile(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return (i, j)

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

    def get_possible_moves(self):
        row, col = self.empty_tile
        possible_moves = []
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # down, up, right, left
        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                # Make the move
                new_board = [row[:] for row in self.board]  # Deep copy
                new_board[row][col], new_board[new_row][new_col] = new_board[new_row][new_col],
new_board[row][col]
                possible_moves.append(PuzzleState(new_board, self.moves + 1, self))
        return possible_moves

def dfs(initial_state, goal_state):
    stack = [initial_state]
    visited = set()
    while stack:
        current_state = stack.pop()
        # If we find the goal, return the state
        if current_state.is_goal(goal_state):
            return current_state
        # Convert board to a tuple for the visited set
```

```
        state_tuple = tuple(tuple(row) for row in current_state.board)
        # If we've already visited this state, skip it
        if state_tuple not in visited:
            visited.add(state_tuple)
            for next_state in current_state.get_possible_moves():
                stack.append(next_state)
    return None  # No solution found

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for state in reversed(path):
        for row in state:
            print(row)
        print()

if __name__ == "__main__":
    # Get user input for initial and goal states
    initial_board = get_user_input("Enter the initial state of the puzzle:")
    goal_board = get_user_input("Enter the goal state of the puzzle:")

    if is_solvable(initial_board):
        initial_state = PuzzleState(initial_board)
        solution = dfs(initial_state, goal_board)
        if solution:
            print("Solution found in", solution.moves, "moves:")
            print_solution(solution)
        else:
            print("No solution found.")
    else:
        print("This puzzle is unsolvable.")
```

Initial State :

1 2 3
4 5 6
0 7 8

1 2 3
0 5 6
4 7 8

---

0 2 3
1 5 6
4 7 8

---

2 0 3
1 5 6
4 7 8

⋮

1 2 3
4 5 6 } final stat
7 8 0

Program 03 - 8 Puzzle Using A*

Algorithm

Lab 04 : A* Search Algorithm

Pseudocode:

```
function A* search(problem) returns a solution or failure
node ← a node n with n.states = 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
    if problem.goalTest(n.State) then return solution(n)
    for each action a in problem.action(n.state) do
            n' ← childNode(problem, n, a)
            insert(n', g(n') + h(n'), frontier)
```

Output: (Manhattan Distance)

Start State:

2 8 3
1 6 4
7 _ 5

Goal State:

1 2 3
8 _ 4
7 6 5

Solution found in 5 moves using manhattan heuristic

2 8 3
1 6 4
7 _ 5
↓
~~Move 1~~
2 8 3

Move 3

_ 2 3
1 8 4
7 6 5

Move 4

1 2 3

Code

MANHATTAN DISTANCE

```
#Manhattan Distance
import heap
class Node:
    def _init_(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.g = 0  # Cost from start to this node
        self.h = 0  # Heuristic cost from this node to target
        self.f = 0  # Total cost

    def _lt_(self, other):
        return self.f < other.f

def heuristic(a, b):
    # Manhattan distance
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar(start, goal, grid):
    open_list = []
    closed_list = set()
    start_node = Node(start)
    goal_node = Node(goal)
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
```

```
    closed_list.add(current_node.position)

    # Goal check
    if current_node.position == goal:
        path = []
        while current_node:
            path.append(current_node.position)
            current_node = current_node.parent
        return path[::-1]  # Return reversed path

    # Generate neighbors
    neighbors = [
        (current_node.position[0] + dx, current_node.position[1] + dy)
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]
    ]

    for next_position in neighbors:
        # Check if within bounds and not a wall (assuming 0 is free space)
        if (0 <= next_position[0] < len(grid) and
            0 <= next_position[1] < len(grid[0]) and
            grid[next_position[0]][next_position[1]] == 0):

            if next_position in closed_list:
                continue

            neighbor_node = Node(next_position, current_node)
            neighbor_node.g = current_node.g + 1
            neighbor_node.h = heuristic(next_position, goal)
            neighbor_node.f = neighbor_node.g + neighbor_node.h

            # Check if this neighbor is already in the open list
            if any(neighbor.position == neighbor_node.position and neighbor.f <= neighbor_node.f for
neighbor in open_list):
                continue

            heapq.heappush(open_list, neighbor_node)
```

```
    return []  # Return empty path if no path found


# Example usage
if _name___== "_main_":
    grid = [
        [0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0],
        [0, 0, 0, 0, 0]
    ]
    start = (0, 0)
    goal = (4, 4)
path = astar(start, goal, grid)
print("Path from start to
goal:", path)
print("Venkatesh Vinay
Chandlle")
print("1BM22CS325")
```

MISPLACED TILES

```
#Misplaced Tiles
```

```
import heapq


class PuzzleState:
    def __init__(self, board, g=0):
```

```
        self.board = board
        self.g = g  # Cost from start to this state
        self.zero_pos = board.index(0)  # Position of the empty space

    def h(self):
        # Calculate the number of misplaced tiles (Misplaced Tile Heuristic)
        return sum(1 for i in range(9) if self.board[i] != 0 and self.board[i] != i + 1)

    def f(self):
        return self.g + self.h()

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, 3)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_zero_pos = new_x * 3 + new_y
                new_board = self.board[:]
                # Swap zero with the neighboring tile
                new_board[self.zero_pos], new_board[new_zero_pos] = new_board[new_zero_pos],
new_board[self.zero_pos]
                neighbors.append(PuzzleState(new_board, self.g + 1))
        return neighbors

def a_star(initial_state, goal_state):
    open_set = []
    heapq.heappush(open_set, (initial_state.f(), 0, initial_state))  # Add a unique identifier (0 in this
case)
    came_from = {}
```

```
    g_score = {tuple(initial_state.board): 0}

    while open_set:
        current_f, _, current = heapq.heappop(open_set)

        if current.board == goal_state:
            return reconstruct_path(came_from, current)

        for neighbor in current.get_neighbors():
            neighbor_tuple = tuple(neighbor.board)
            tentative_g_score = g_score[tuple(current.board)] + 1

            if neighbor_tuple not in g_score or tentative_g_score < g_score[neighbor_tuple]:
                came_from[neighbor_tuple] = current
                g_score[neighbor_tuple] = tentative_g_score
                heapq.heappush(open_set, (neighbor.f(), neighbor.g, neighbor))

    return None  # No solution found

def reconstruct_path(came_from, current):
    path = []
    while current is not None:
        path.append(current.board)
        current = came_from.get(tuple(current.board), None)
    return path[::-1]

# Example usage
if __name__ == "__main__":
    initial_state = PuzzleState([1, 2, 3, 4, 5, 6, 0, 7, 8])
    goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
```

```
    solution = a_star(initial_state, goal_state)


    if solution:
        for step in solution:
            print(step)
    else:
        print("No solution found")

print("Venkatesh Vinay
Chandlle")
print("1BM22CS325")
```

Misplaced Tiles

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

(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, f(n)=4 | h(n)=5, f(n)=6 | h(n)=5, f(n)=6

g(n)=2

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

h(n) = 3
f(n) = 5

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
f(n) = 5

g(n) = 4

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

h(n)=1
f(n) = 5

f(n) = 5

Program 4 - Vacuum Cleaner

Algorithm

Lab-02

2. Implment Vaccum Cleaner Agent

Pseudocode

Function vaccum_world(){
    initialize goal_state = {'A':'0', 'B':'0'}
    initialize cost = 0
    Input location
    Input status for location
    Input status for other location
    Print Initial condition for Location, goal state
If location input = 'A' and status_input = '1' then, {
    print location A is dirty
    goal_state['A'] = '0'
    cost += 1
    print cost for ~~the~~ cleaning 'A' - cost
If status for other location = '1' then {
    print location B is Dirty
    cost += 1
    print cost for moving right as cost }
~~print "Vaccum is placed in Location B"~~
If status_input == 1 then Location B is dirty
else {
    print Location A is already clean
    If status of other location = 1 then {
        print Location B is dirty
        cost += 1
        Print cost for moving right = cost
        cost += 1
        Print total cost of cleaning, cost
}}

Else { ~~print~~ print vaccum is at Location B }
    If status == '1' then print 'Location B is dirty'
    cost = 1
    cost for cleaning, cost
    If status of other location = 1, then {
        print Location = 1 then,
        cost += 1 ~~print~~
        print cost for moving Left, cost
        goal state['A'] = '0'
        cost += 1
        print cost for cleaning, cost }
    else {
        print location B is already clean
        If status other location = '1' then {
            print Location A is dirty
            cost += 1
            print cost for moving Left, cost
            goal state['A'] = '0'
            cost += 1
            print cost for clean, cost
    }
    print performance Measurement cost.
}

Output

Locations: A-0 B-1
Enter Location of vaccum: 1 (at B)
Enter status of Room (0 for clean, 1 for dirty): 1
Enter status of other room (0 for clean, 1 for dirty): 0
Initial location condition
Vaccum is placed in B
Location B is Dirty
Cost for cleaning: 1

Code

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum: ")
    status_input = input("Enter status of " + location_input + " (0 for Clean, 1 for Dirty): ")
    status_input_complement = input("Enter status of other room (0 for Clean, 1 for Dirty): ")

    print("Initial Location Condition: " + str(goal_state))

    if location_input == 'A':
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            goal_state['A'] = '0'
            cost += 1
            print("Cost for CLEANING A: " + str(cost))
            print("Location A has been Cleaned.")

        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving right to Location B.")
            cost += 1
            print("COST for moving RIGHT: " + str(cost))
            goal_state['B'] = '0'
            cost += 1
            print("COST for SUCK: " + str(cost))
            print("Location B has been Cleaned.")
        else:
            print("No action needed; Location B is already clean.")
    else:
        print("Location A is already clean.")
        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving RIGHT to Location B.")
```

```
        cost += 1
        print("COST for moving RIGHT: " + str(cost))
        goal_state['B'] = '0'
        cost += 1
        print("COST for SUCK: " + str(cost))
        print("Location B has been Cleaned.")
    else:
        print("No action needed; Location B is already clean.")

if location_input == 'B':
    print("Vacuum is placed in Location B")
    if status_input == '1':
        print("Location B is Dirty.")
        goal_state['B'] = '0'
        cost += 1
        print("COST for CLEANING B: " + str(cost))
        print("Location B has been Cleaned.")

    if status_input_complement == '1':
        print("Location A is Dirty.")
        print("Moving LEFT to Location A.")
        cost += 1
        print("COST for moving LEFT: " + str(cost))
        goal_state['A'] = '0'
        cost += 1
        print("COST for SUCK: " + str(cost))
        print("Location A has been Cleaned.")
    else:
        print("No action needed; Location A is already clean.")
else:
    print("Location B is already clean.")
    if status_input_complement == '1':
        print("Location A is Dirty.")
        print("Moving LEFT to Location A.")
        cost += 1
        print("COST for moving LEFT: " + str(cost))
        goal_state['A'] = '0'
```

```
        cost += 1
        print("COST for SUCK: " + str(cost))
        print("Location A has been Cleaned.")
    else:
        print("No action needed; Location A is already clean.")

  print("GOAL STATE: ")
  print(goal_state)
print("Performance
Measurement: " + str(cost))
print("Venkatesh Vinay
Chandlle")
print("1BM22CS325")



vacuum_world()
```

Program-05 Hill Climbing

Algorithm

N-Queen Implementation Using hill-Climbing algorithm

Algorithm for hill Climbing Algorithm

function Hill-Climbing (Problem) returns a state that is lo ~~maximum~~
maximum

current ← Make-Node (Problem, Initial-State)

loop do

neighbour ← a highest valued function

if neighbour.value ≤ current.value
then return state

end if

current ← neighbour

end loop.

Executed

Code

```
import random

def print_board(board, n):
    """Prints the current state of the board."""
    for row in range(n):
        line = ""
        for col in range(n):
            if board[col] == row:
                line += " Q "
            else:
                line += " . "
        print(line)
    print()

def calculate_conflicts(board, n):
    """Calculates the number of conflicts (attacks) between queens."""
    conflicts = 0
    for i in range(n):
        for j in range(i + 1, n):
            # Check if queens are in the same row or diagonal
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def get_best_neighbor(board, n):
    """
    Finds the best neighboring board with the fewest conflicts.
    Returns the best board and its conflict count.
    """
    current_conflicts = calculate_conflicts(board, n)
    best_board = board[:]
    best_conflicts = current_conflicts
    neighbors = []

    for col in range(n):
        original_row = board[col]
```

```
    for row in range(n):
      if row == original_row:
        continue
      # Move queen to a new row and calculate conflicts
      board[col] = row
      new_conflicts = calculate_conflicts(board, n)
      neighbors.append((board[:], new_conflicts))
    # Restore the original row before moving to the next column
    board[col] = original_row

  # Sort neighbors by the number of conflicts (ascending)
  neighbors.sort(key=lambda x: x[1])
  if neighbors:
    best_neighbor = neighbors[0]
    if best_neighbor[1] < best_conflicts:
      return best_neighbor
  return board, current_conflicts

def hill_climbing_with_restarts(n, initial_board, max_restarts=100):
  """
  Performs Hill Climbing with random restarts to solve the N-Queens problem.
  Returns the final board configuration and its conflict count.
  """
  current_board = initial_board[:]
  current_conflicts = calculate_conflicts(current_board, n)

  print("Initial board:")
  print_board(current_board, n)
  print(f"Initial conflicts: {current_conflicts}\n")

  steps = 0
  restarts = 0

  while current_conflicts > 0 and restarts < max_restarts:
    new_board, new_conflicts = get_best_neighbor(current_board, n)

    steps += 1
```

```
        print(f"Step {steps}:")
        print_board(new_board, n)
        print(f"Conflicts: {new_conflicts}\n")

        if new_conflicts < current_conflicts:
            current_board = new_board
            current_conflicts = new_conflicts
        else:
            # If no better neighbor is found, perform a random restart
            restarts += 1
            print(f"Restarting... (Restart number {restarts})\n")
            current_board = [random.randint(0, n-1) for _ in range(n)]
            current_conflicts = calculate_conflicts(current_board, n)
            print("New initial board:")
            print_board(current_board, n)
            print(f"Conflicts: {current_conflicts}\n")

    return current_board, current_conflicts

# Main function
def main():
    n = 4
    print("Enter the initial positions of queens (row numbers from 0 to 3 for each column):")
    initial_board = []
    for i in range(n):
        while True:
            try:
                row = int(input(f"Column {i}: "))
                if 0 <= row < n:
                    initial_board.append(row)
                    break
                else:
                    print(f"Please enter a number between 0 and {n-1}.")
            except ValueError:
                print("Invalid input. Please enter an integer.")

    solution, conflicts = hill_climbing_with_restarts(n, initial_board)
```

```
    print("Final solution:")
    print_board(solution, n)
    if conflicts == 0:
        print("A solution was found with no conflicts!")
    else:
        print(f"No solution was found after {100} restarts. Final number of conflicts: {conflicts}")

if _name___== "_main_":
    main()
print("Venkatesh Vinay
Chandlle")
print("1BM22CS325")
```