

# LABORATORY PROGRAM – 1

## IMPLEMENT TIC –TAC –TOE GAME

### PSEUDOCODE OR ALGORITHM

10.24

DATE: PAGE:

LABORATORY PROGRAM -1

1. Implement Tic-Tac-Toe Game.

→ Algorithm :

1. Initialize the game board, creating a  $3 \times 3$  board represented as a list of 9 elements.
2. Display the board, print the current state of the board.
3. Check for Win; evaluate all winning combinations (rows, columns, diagonals). Return the winning player or none.
4. Check for Tie; if there are no empty spaces and no winner, return True.
5. Main Game Loop;  
Set the current player to "X".  
while the game is not over:  
Display the board.  
Prompt the current player for their move (or make a random move for the computer).  
Update the board.  
Check for a winner or a tie.  
Switch the current player.  
End game;  
Announce the result (win or tie).

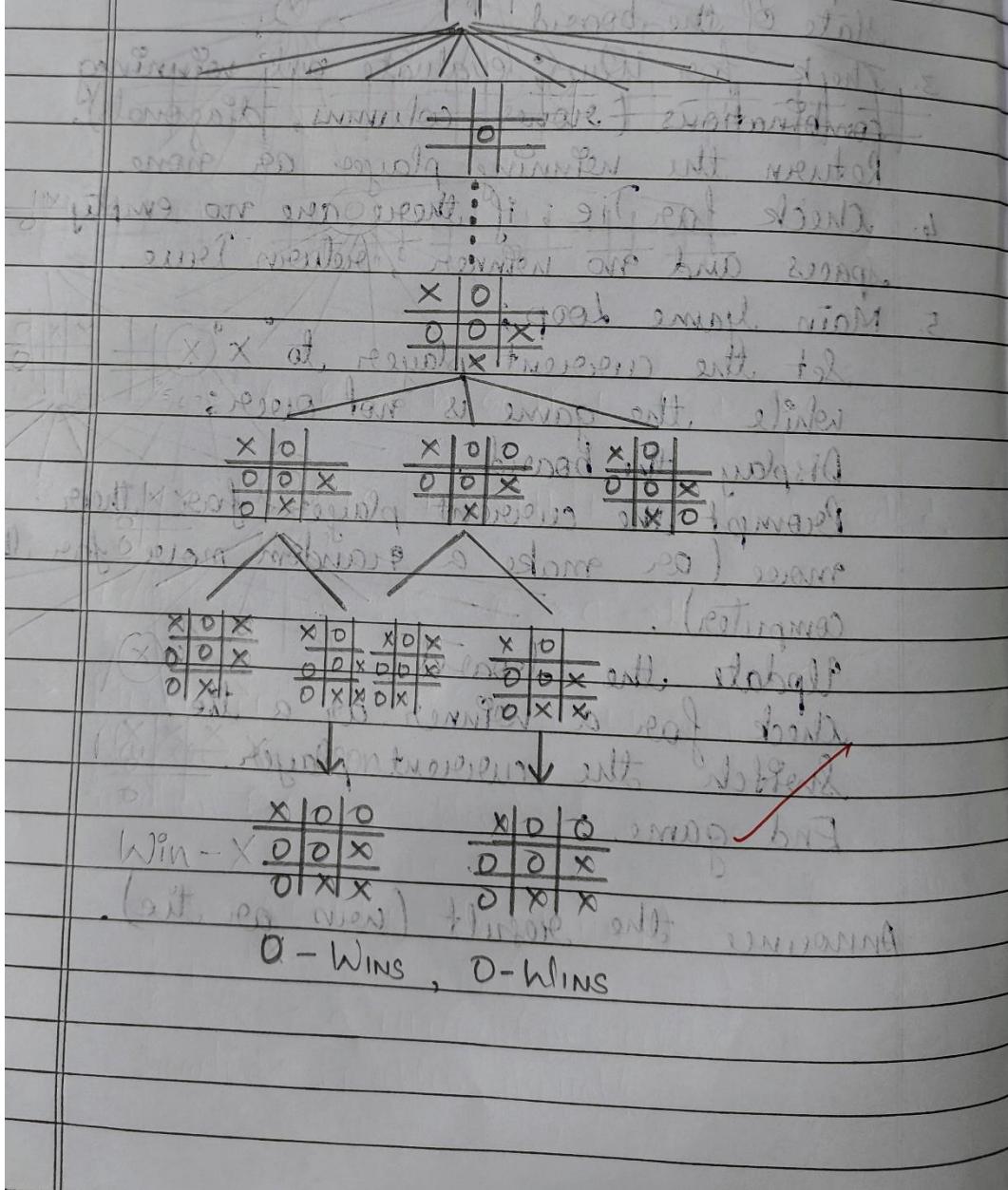
# STATE SPACE TREE

DATE:

PAGE:

↳ STATE SPACE TREE:

1. Root Node: Represents the empty board.
2. Level 1: Player X's moves (9 possible positions)
3. Level 2: Computer's move (varying no. of position)
4. Level 3: Player X's subsequent moves, continuing until a win or tie is reached.



## CODE

```
import random

board = ["-", "-", "-",
        "-", "-", "-",
        "-", "-", "-"]

def p_board(player):
    print(board[0] + " | " + board[1] + " | " + board[2])
    print(board[3] + " | " + board[4] + " | " + board[5])
    print(board[6] + " | " + board[7] + " | " + board[8])

def check_win():
    for i in range(0, 7, 3):
        if board[i] == board[i + 1] == board[i + 2] and
           board[i] != '-':
            return board[i]

    for i in range(3):
        if board[i] == board[i + 3] == board[i + 6] and
           board[i] != '-':
            return board[i]

    if board[0] == board[4] == board[8] and board[0] != '-':
        return board[0]
    if board[2] == board[4] == board[6] and board[2] != '-':
        return board[2]

    return None

def check_tie():
    if '-' not in board:
        return True
    return False

def play_game():
    current_player = "X"
    game_over = False

    while not game_over:
        p_board(current_player)

        if current_player == "X":
            print("It's your turn (X).")
            try:
                position = int(input("Choose a position from 1-9:"))
            except ValueError:
                print("Invalid input. Please enter a number between 1 and 9.")
                continue
            if position < 0 or position > 8 or board[position] != '-':
                print("Invalid move. Try again.")
                continue
            else:
                print("Computer's turn (O).")
                available_moves = [i for i, spot in
                                   enumerate(board) if spot == '-']
                position = random.choice(available_moves)

        board[position] = current_player
        winner = check_win()

        if winner:
            p_board(current_player)
            print(winner + " won!")
            game_over = True
        elif check_tie():
            p_board(current_player)
            print("It's a tie!")
            game_over = True
        else:
            current_player = "O" if current_player == "X" else "X"

        play_game()
```

## OUTPUT

```
- | - | -
- | - | -
- | - | -
It's your turn (X).
Choose a position from 1-9: 5
- | - | -
- | X | -
- | - | -
Computer's turn (O).
- | O | -
- | X | -
- | - | -
It's your turn (X).
Choose a position from 1-9: 7
- | O | -
- | X | -
X | - | -
Computer's turn (O).
- | O | O
- | X | -
X | - | -
It's your turn (X).
Choose a position from 1-9: 1
X | O | O
- | X | -
X | - | -
Computer's turn (O).
X | O | O
- | X | O
X | - | -
It's your turn (X).
Choose a position from 1-9: 4
X | O | O
X | X | O
X | - | -
X won!
```

# LABORATORY PROGRAM – 2

## SOLVE 8 PUZZLE PROBLEMS

### PSEUDOCODE OR ALGORITHM

8.10.24.

DATE: PAGE:

LABORATORY PROGRAM 13-22(B)

1. Pseudocode for 8 puzzle problem:

```
function FIND-ZERO(state) returns (row, col)
    for i from 0 to 2 do
        for j from 0 to 2 do
            if state[i][j] = 0 then
                return (i, j)
```

function MOVE (state, direction) return new-state
 new-state ← copy of state with ref.
 (i, j) ← FIND-ZERO(state)
 if direction = "up" and i > 0 then
 swap new-state[i][j] with new-state[i-1][j]
 else if direction = "down" and i < 2 then
 swap new-state[i][j] with new-state[i+1][j]
 else if direction = "left" and j > 0 then
 swap new-state[i][j] with new-state[i][j-1]
 else if direction = "right" and j < 2 then
 swap new-state[i][j] with new-state[i][j+1].
 return new-state at 0 mark & ref.

function IS-GOAL(state) returns boolean
 return state == goal-state

function PRINT-STATE(state)
 for each row in state do
 print row.

```

function DFS(initial_state) {
    stack ← [(initial_state, [])]
    visited ← empty set
    while not is-empty(stack) do
        (state, path) ← pop(stack)
        PRINT-STATE(state)
        if IS-GOAL(state) then
            return path
        else
            for direction in ["up", "down", "left", "right"] do
                new_state ← move(state, direction)
                if new_state is not null and str(new_state) not
                    in visited then
                        push(stack, [new_state, path + [direction]])
    return failure
}

function GET-INITIAL-STATE() {
    print "Enter the initial state of the 8-puzzle"
    if (row ← input("row")) is not null then
        initial_state ← empty list
        for i from 0 to 2 do
            row ← input("row") as list of integers
            initial_state.append(row)
        return initial_state
    else
        print "Input error"
}

main()
initial_state ← GET-INITIAL-STATE()
if PRINT-STATE(initial_state) is not null then
    start_time ← current_time
    print "Solving using DFS:"
    dfs_solution ← DFS(initial_state)
    end_time ← current_time
    if dfs_solution is not null then
        print "DFS Solution:", dfs_solution
    else
        print "No solution found."
    print "Time taken by DFS:", end_time - start_time
}

```

DATE: PAGE:

$\checkmark$    
 dfs\_solution ← DFS(initial\_state);   
 end\_time ← current\_time   
 if dfs\_solution is not null then   
 print "DFS Solution:", dfs\_solution.   
 else   
 print "No solution found."   
 print "Time taken by DFS:", end\_time - start\_time

# STATE SPACE TREE

8.10.24

DATE:

PAGE:

LABORATORY PROGRAM + 2(A) 04/13/21

STATE SPACE TREE FOR EIGHT PUZZLE PROBLEM USING DFS.

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

Left	Right
1 2 3	1 2 3
4 0 6	4 5 6
7 5 8	7 0 8

Goal Reached.  
DFS Solution: ['right']

## CODE

```
from collections import deque
import time

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

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

def move(state, direction):
    new_state = [row[:] for row in state]
    zero_pos = find_zero(state)
    i, j = zero_pos

    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
    elif direction == "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 is_goal(state):
    return state == goal_state

def print_state(state):
    for row in state:
        print(row)
    print("\n")

def dfs(initial_state):
    stack = [(initial_state, [])]
    visited = set()

    while stack:
        state, path = stack.pop()
        print("Exploring state in DFS:")
        print_state(state)

        if is_goal(state):
            return path

        visited.add(str(state))
        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and str(new_state) not in visited:
                stack.append((new_state, path + [direction]))

    return None

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()

    print("Initial State:")
    print_state(initial_state)

    start_time_dfs = time.time()
    print("Solving using DFS:")
    dfs_solution = dfs(initial_state)
    end_time_dfs = time.time()
    if dfs_solution:
        print("DFS Solution:", dfs_solution)
    else:
        print("No solution found with DFS.")
        print(f"Time taken by DFS: {end_time_dfs - start_time_dfs:.6f} seconds")
```

## OUTPUT

```
Enter the initial state of the 8-puzzle (0 for empty space):  
Enter row 1 (space-separated): 1 2 3  
Enter row 2 (space-separated): 4 6 5  
Enter row 3 (space-separated): 8 7 0  
Initial State:  
[1, 2, 3]  
[4, 6, 5]  
[8, 7, 0]  
  
Solving using DFS:  
Exploring state in DFS:  
[1, 2, 3]  
[4, 6, 5]  
[8, 7, 0]  
  
Exploring state in DFS:  
[1, 2, 3]  
[4, 6, 5]  
[8, 0, 7]  
  
Exploring state in DFS:  
[1, 2, 3]  
[0, 6, 5]  
[4, 8, 7]  
  
Exploring state in DFS:  
[1, 2, 3]  
[4, 8, 5]  
[7, 0, 6]  
  
Exploring state in DFS:  
[1, 2, 3]  
[4, 8, 5]  
[7, 6, 0]  
  
Exploring state in DFS:  
[1, 2, 3]  
[4, 0, 5]  
[7, 8, 6]  
  
Exploring state in DFS:  
[1, 2, 3]  
[4, 5, 0]  
[7, 8, 6]  
  
Exploring state in DFS:  
[1, 2, 3]  
[4, 5, 6]  
[7, 8, 0]
```

# LABORATORY PROGRAM – 3

## IMPLEMENT ITERATIVE DEEPENING SEARCH ALGORITHM

### PSEUDOCODE OR ALGORITHM

2 Pseudocode for iterative deepening search algorithm.

class NODE

function INIT(state, parent)

self.state ← state

self.parent ← parent

function PATH() returns list

node ← self.

result ← empty list

while node is not null do

append node.state to result

node ← node.parent

return REVERSE(result)

function I-D-S(problem) returns path

depth ← 0

while true do

print "Exploring depth:", depth,

(result, \_) ← D-F-S (problem, depth)

if result is not null and result is not "cutoff" then

return result

depth ← depth + 1.

function D-F-S(problem, limit) returns path or integer  
 frontier  $\leftarrow$  [node(problem, initial-state)]  
 explored  $\leftarrow$  empty-set  
 cutoff-occurred  $\leftarrow$  false  
 while frontier is not empty do  
     node  $\leftarrow$  pop(frontier)  
     if problem.IS-GOAL(node.state) then  
         return node.PATH(), explored.  
     if node.state not in explored then  
         explored.add(node.state)  
         if LENGTH(node.PATH()) - 1 < limit then  
             for child in problem.EXPAND(node.state) do  
                 APPENDS(frontier, node.PATH() + child.state)  
     else  
         cutoff-occurred  $\leftarrow$  true  
 return "cutoff" if cutoff-occurred else null, explored

### class GRAPH-PROBLEM

function INIT(initial-state, goal-state, adjacency-list)  
 self.initial-state  $\leftarrow$  initial-state  
 self.goal-state  $\leftarrow$  goal-state  
 self.adjacency-list  $\leftarrow$  adjacency-list

function IS-GOAL(state) returns boolean  
 return state == self.goal-state

function EXPAND(state) returns list,  
 returns [neighbor for neighbor in self.adjacency  
 list.get(state, [ ])].

```

function GET-GRAPH-FROM-INPUT() returns GRAPH-PROBLEM
    adjacency-list <- empty-dictionary
    initial-state <- INPUT ("Enter the initial state : ").strip()
    goal-state <- INPUT ("Enter the goal state : ").strip()
    print "Enter the AD list for graph"
    print "Type 'done' when finished."
    while true do
        node <- INPUT ("Enter node : ").strip()
        if node.lower() == "done" then
            break
        neighbors-input <- PINPUT ("Enter N of " + node + " separated by spaces : ").strip()
        neighbors <- neighbors-input.split()
        adjacency-list [node] <- [neighbor.strip() for neighbor in neighbors]
    return GRAPH-PROBLEM (initial-state, goal-state,
                           adjacency-list)

```

mais

problem ← GET - GRAPH - FROM - INPUT()

solution ← I-D-S(problem)

If solution is not null then

point "solution Path:", solution  
else

point "No solution found".

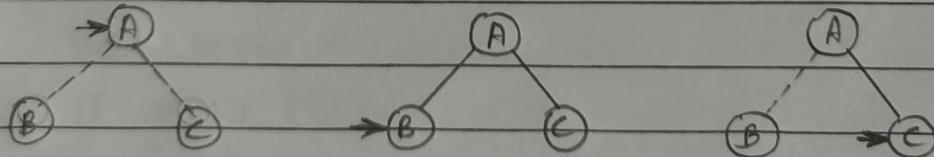
## STATE SPACE TREE

↳ STATE SPACE TREE FOR ITERATIVE DEEPENING SEARCH ALGORITHM

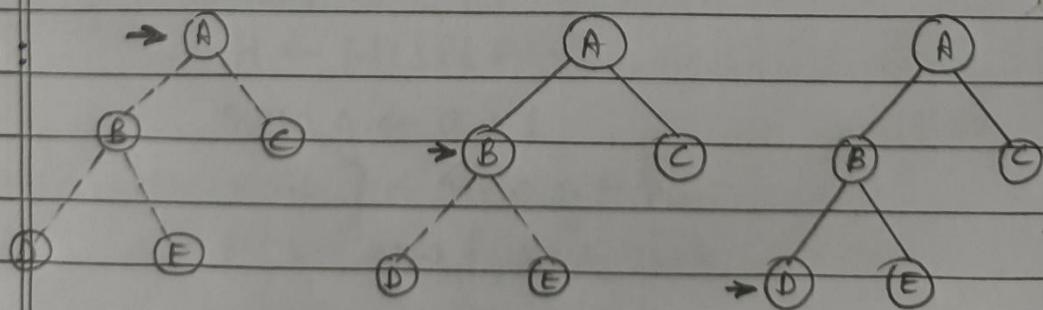
limit 0 : → A

INITIAL STATE : A , GOAL STATE : D

limit 1 :



limit 2 :



Goal Reached

Solution Path : ['A', 'B', 'D']

## CODE

```
class Node:  
    def __init__(self, state, parent=None):  
        self.state = state  
        self.parent = parent  
  
    def path(self):  
        node, result = self, []  
        while node:  
            result.append(node.state)  
            node = node.parent  
        return result[::-1]  
  
def iterative_deepening_search(problem):  
    depth = 0  
    while True:  
        print(f"Exploring depth: {depth}")  
        result, _ = depth_limited_search(problem, depth)  
        if result is not None and result != 'cutoff':  
            return result  
        depth += 1  
  
def depth_limited_search(problem, limit):  
    frontier = [Node(problem.initial_state)]  
    explored = set()  
    cutoff_occurred = False  
  
    while frontier:  
        node = frontier.pop()  
  
        if problem.is_goal(node.state):  
            return node.path(), explored  
  
        if node.state not in explored:  
            explored.add(node.state)  
            if len(node.path()) - 1 < limit:  
                for child in problem.expand(node.state):  
                    frontier.append(Node(child, node))  
            else:  
                cutoff_occurred = True  
  
    return 'cutoff' if cutoff_occurred else None, explored  
  
class GraphProblem:  
  
    def __init__(self, initial_state, goal_state,  
                 adjacency_list):  
        self.initial_state = initial_state  
        self.goal_state = goal_state  
        self.adjacency_list = adjacency_list  
  
    def is_goal(self, state):  
        return state == self.goal_state  
  
    def expand(self, state):  
        return [neighbor for neighbor in  
                self.adjacency_list.get(state, [])]  
  
    def get_graph_from_input():  
        adjacency_list = {}  
        initial_state = input("Enter the initial state: ").strip()  
        goal_state = input("Enter the goal state: ").strip()  
  
        print("Enter the adjacency list for the graph  
(neighbors of each node).")  
        print("Type 'done' when finished.")  
  
        while True:  
            node = input("Enter node (or 'done' to finish):  
").strip()  
            if node.lower() == 'done':  
                break  
            neighbors_input = input(f"Enter neighbors of  
{node} separated by spaces: ").strip()  
            neighbors = neighbors_input.split()  
            adjacency_list[node] = [neighbor.strip() for  
                                   neighbor in neighbors]  
  
        return GraphProblem(initial_state, goal_state,  
                            adjacency_list)  
  
    if __name__ == "__main__":  
        problem = get_graph_from_input()  
        solution = iterative_deepening_search(problem)  
  
        if solution:  
            print("Solution Path:", solution)  
        else:  
            print("No solution found.")
```

## OUTPUT

```
Enter the initial state: A
Enter the goal state: G
Enter the adjacency list for the graph (neighbors of each node).
Type 'done' when finished.
Enter node (or 'done' to finish): A
Enter neighbors of A separated by spaces: B C
Enter node (or 'done' to finish): B
Enter neighbors of B separated by spaces: D E
Enter node (or 'done' to finish): D
Enter neighbors of D separated by spaces: H I
Enter node (or 'done' to finish): C
Enter neighbors of C separated by spaces: F G
Enter node (or 'done' to finish): F
Enter neighbors of F separated by spaces: K
Enter node (or 'done' to finish): done
Exploring depth: 0
Exploring depth: 1
Exploring depth: 2
Solution Path: ['A', 'C', 'G']
```

# LABORATORY PROGRAM – 4

## IMPLEMENT VACUUM CLEANER AGENT

### PSEUDOCODE OR ALGORITHM

DATE: PAGE: 12.01.8

2. Implement vacuum cleaner agent.

→ Algorithm:

1. Initialize:  
Set a 2D grid (status) indicating cleanliness.  
Start at the initial location (0,0).

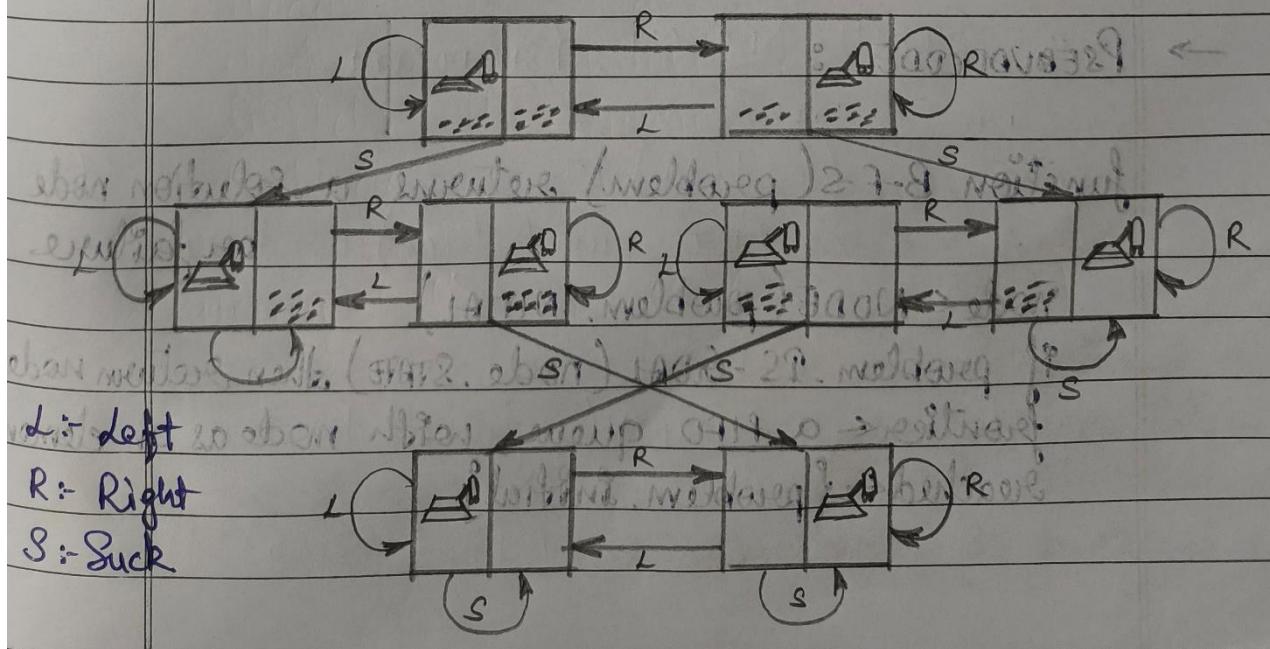
2. Agent Function:  
Define vacuum cleaner agent(location, status):  
Check if the current location is 'Dirty' or 'Clean'.  
Return the appropriate action message.

3. Main Loop:  
Repeat until all locations are clean:  
Get the action from the agent function  
and print it.  
If the location is 'Dirty', clean it.  
Check if all locations are clean; if so,  
print completion message and exit.  
Update the location (move right, then down)

*Sun 11/01/24*

STATE SPACE TREE

$\Rightarrow$  STATE SPACE DIAGRAM :



## CODE

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter the initial location of
the vacuum cleaner (A or B): ")
    status_input = input(f"Enter the status of room
{location_input} (0 for Clean, 1 for Dirty): ")
    status_input_complement = input(f"Enter the status
of the other room ({'B' if location_input == 'A' else
'A'}) (0 for Clean, 1 for Dirty): ")

    # Set the initial state based on user input
    initial_state = {
        'A': status_input if location_input == 'A' else
status_input_complement,
        'B': status_input_complement if location_input ==
'A' else status_input
    }

    print("\nInitial Location Condition:", initial_state)

    if location_input == 'A':
        print("\nVacuum cleaner is placed in room A.")

        if status_input == '1':
            print("Room A is Dirty. Cleaning room A...")
            goal_state['A'] = '0'
            cost += 1
            print("Cost for cleaning room A:", cost)

        if status_input_complement == '1':
            print("\nRoom B is also Dirty. Moving to
room B...")
            cost += 1
            print("Cost for moving to room B:", cost)
            print("Cleaning room B...")
            goal_state['B'] = '0'
            cost += 1
            print("Cost for cleaning room B:", cost)
            else:
                print("\nRoom B is already Clean. No further
action needed.")

        else:
            print("Room A is already Clean.")

            if status_input_complement == '1':
                print("\nRoom B is Dirty. Moving to room
B...")
                cost += 1
                print("Cost for moving to room B:", cost)
                print("Cleaning room B...")
                goal_state['B'] = '0'
                cost += 1
                print("Cost for cleaning room B:", cost)
                else:
                    print("\nRoom A is already Clean. No further
action needed.")

    print("\nGOAL STATE:", goal_state)
    print("Total cost for cleaning:", cost)

vacuum_world()
```

## OUTPUT

```
Enter the initial location of the vacuum cleaner (A or B): A
Enter the status of room A (0 for Clean, 1 for Dirty): 1
Enter the status of the other room (B) (0 for Clean, 1 for Dirty): 1

Initial Location Condition: {'A': '1', 'B': '1'}

Vacuum cleaner is placed in room A.
Room A is Dirty. Cleaning room A...
Cost for cleaning room A: 1

Room B is also Dirty. Moving to room B...
Cost for moving to room B: 2
Cleaning room B...
Cost for cleaning room B: 3

GOAL STATE: {'A': '0', 'B': '0'}
Total cost for cleaning: 3
```

# LABORATORY PROGRAM – 5(A)

## IMPLEMENT A\* SEARCH ALGORITHM

### PSEUDOCODE OR ALGORITHM

PAGE: 11

ms PSEUDO CODE , A\* IMPLEMENTATION , NUMBER OF MISPLACED TILES :

FUNCTION MISPLACED-TILES(state) RETURNS count  
Count ← 0  
FOR i FROM 0 to 2 DO  
  FOR j from 0 to 2 DO  
    IF state[i][j] = goalState[i][j] AND state[i][j] ≠ 0,  
      Count += 1  
RETURN count

function A-STAR(initial-state) returns path or failure  
pq ← EMPTY min-heap.  
push(0, initial-state, [ ], 0) onto pq.  
visited ← EMPTY SET  
while pq IS NOT EMPTY DO  
  (f, state, path, g) ← POP FROM pq  
  PRINT-STATE(state)  
  IF isGoal(state) THEN  
    RETURN path.  
  visited add {TO-TUPLE(state)}  
  FOR EACH direction IN ["up", "down", "left", "right"] DO  
    new-state ← move(state, direction)  
    IF new-state IS NOT null AND TO-tuple(new-state) NOT IN VISITED THEN  
      H ← MISPLACED-TILES(new-state)  
      new-g ← g + 1  
      new-f ← new-g + h  
      push(new-f, new-state, path + [direction],  
          new-g) onto pq

Return failure.

→ PSEUDO CODE, A\* IMPLEMENTATION, MANHATTAN DISTANCE

function MANHATTAN-DISTANCE(state) returns distance

```

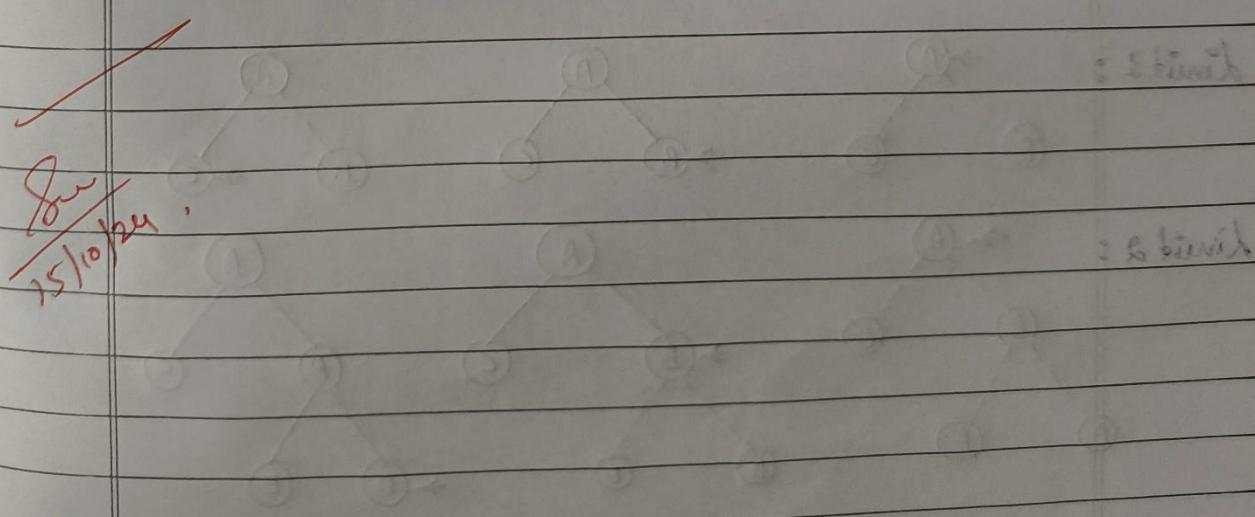
    Distance ← 0
    for i from 0 to 2 do
        for j from 0 to 2 do
            if state[i][j] != 0 then
                goal_i, goal_j ← position of state[i][j] in goal state
                if state[i][j] == 8 then
                    goal_i, goal_j ← 1, 1
                Distance += ABS(goal_i - i) + ABS(goal_j - j)
    return distance
  
```

function A-STAR(initial-state) returns path or failure

Similar to the pseudocode of  
Misplaced tiles, but change in heuristic function

$h \leftarrow \text{MANHATTAN-DISTANCE(state)}$

Rest continues the same



closed loop

['a', 'b', 'c'] = start visited

# STATE SPACE TREE

15.10.24

DATE:

PAGE:

LABORATORY PROGRAM - 3 (A)

→ IMPLEMENT A\* SEARCH ALGORITHM

1. Heuristic : Number filled out of place.

Initial State

1 2 3

8 4

7 6 5

2 8 3

1 6 4

7 5

$h = 5$

$f = 0 + 5 = 5$

Goal State

7 5

$h = 4$

$f = 1 + 4 = 5$

7 5

2 8 3

3 6 4

1 7 5

$h = 5$

$f = 1 + 5 = 6$

$f = 2 + 5 = 7$

2 8 3

2 6 4

7 5

2 8 3

1 6 4

7 5

$f = 3 + 5 = 8$

$f = 2 + 3 = 5$

$f = 3 + 3 = 6$

2 8 3

3 4

7 6 5

2 8 3

1 6 4

7 5

$f = 6$

$f = 4$

$f = 7$

$f = 7$

8 3

2 1 0

7 6 5

$f = 8$

2 8 3

7 3 4

6 5

$f = 5$

2 3

1 8 4

7 6 5

$f = 8$

2 3

1 8 4

7 6 5

$f = 5 + 1 = 6$

1 2 3

8 9

7 6 5

$f = 5 + 3 = 8$

2 3

1 8 4

7 6 5

$f = 8$

1 2 3

8 4

6 5

2 3

1 8 4

7 6 5

$f = 6$

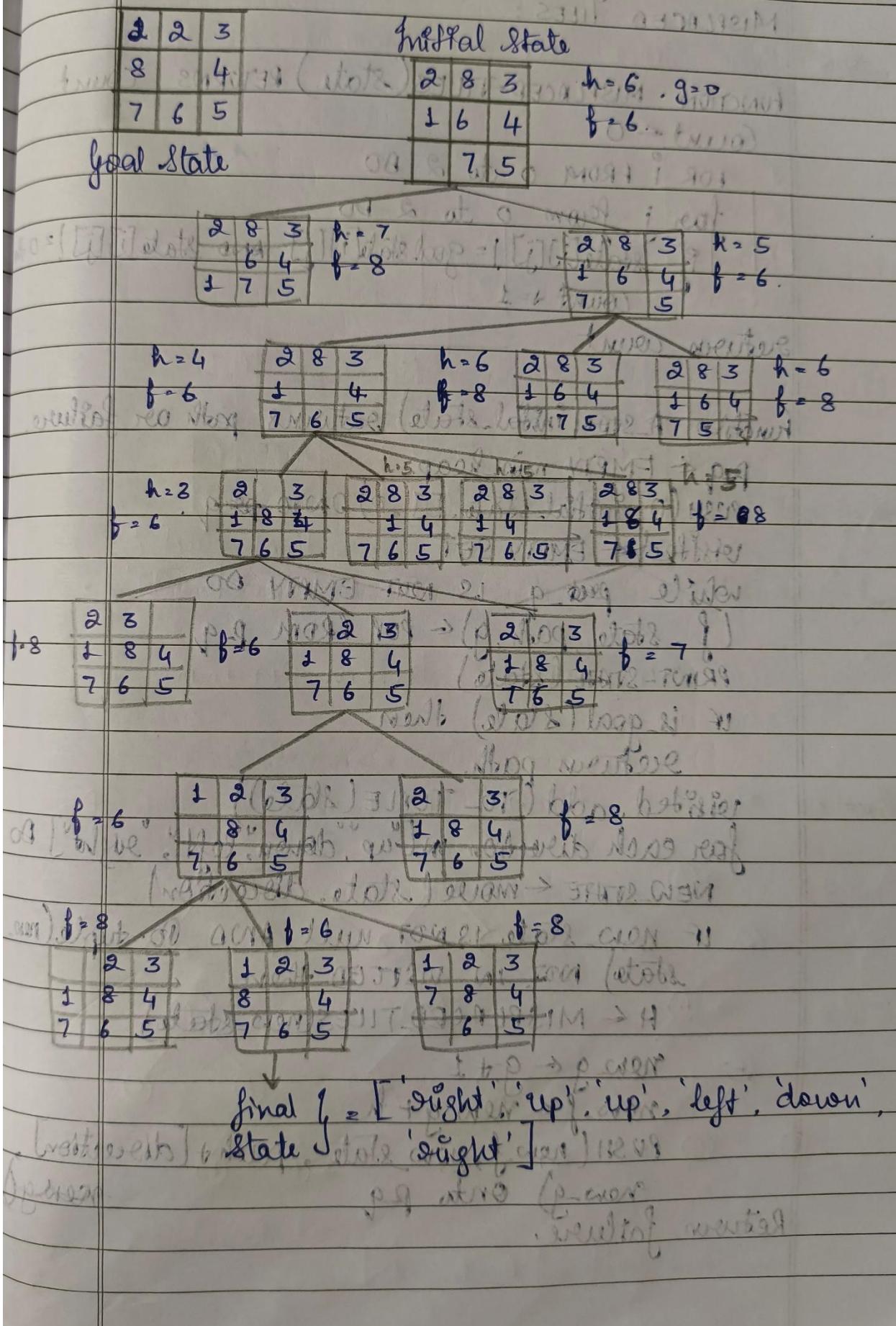
$f = 9$

Final

State

$f = [ \text{'right'}, \text{'up'}, \text{'up'}, \text{'left'}, \text{'down'}, \text{'right'} ]$

2.0 Heuristic : Manhattan Distance



## CODE - G(N): DEPTH OF THE NODE, H(N): NUMBER OF MISPLACED TILES

```
import heapq

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

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

def move(state, direction):
    new_state = [row[:] for row in state]
    i, j = find_zero(state)
    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
    elif direction == "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 is_goal(state):
    return state == goal_state

def misplaced_tiles(state):
    return sum(1 for i in range(3) for j in range(3)
              if state[i][j] != goal_state[i][j] and state[i][j]
              != 0)

def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [], 0))
    visited = set()

    while priority_queue:
        f, state, path, g = heapq.heappop(priority_queue)
        print("Exploring state in A*:")
        print_state(state)

        if is_goal(state):
            return path

        visited.add(tuple(map(tuple, state)))

        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and tuple(map(tuple, new_state)) not in visited:
                h = misplaced_tiles(new_state)
                new_g = g + 1
                new_f = new_g + h
                heapq.heappush(priority_queue, (new_f, new_state, path + [direction], new_g))

    return None

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

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split())))
        if len(row) != 3:
            raise ValueError("Each row must contain exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()
    print("Initial State:")
    print_state(initial_state)
    print("Solving using A* search:")
    a_star_solution = a_star(initial_state)

    if a_star_solution:
        print("A* Solution:", a_star_solution)
    else:
        print("No solution found with A*.")


```

## OUTPUT - G(N): DEPTH OF THE NODE, H(N): NUMBER OF MISPLACED TILES

```
Enter the initial state of the 8-puzzle (0 for empty space):
Enter row 1 (space-separated): 2 8 3
Enter row 2 (space-separated): 1 6 4
Enter row 3 (space-separated): 0 7 5
Initial State:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Solving using A* search:
Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Exploring state in A*:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

Exploring state in A*:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Exploring state in A*:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

Exploring state in A*:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

Exploring state in A*:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

A* Solution: ['right', 'up', 'up', 'left', 'down', 'right']
```

## CODE - G(N): DEPTH OF THE NODE, H(N): MANHATTAN DISTANCE

```
import heapq

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

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

def move(state, direction):
    new_state = [row[:] for row in state]
    i, j = find_zero(state)
    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
    elif direction == "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 is_goal(state):
    return state == goal_state

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i, goal_j = divmod(state[i][j] - 1, 3)
                if state[i][j] == 8:
                    goal_i, goal_j = 1, 1
                distance += abs(goal_i - i) + abs(goal_j - j)
    return distance

def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [], 0))
    visited = set()

    while priority_queue:
        f, state, path, g = heapq.heappop(priority_queue)
        print("Exploring state in A*:")
        print_state(state)

        if is_goal(state):
            return path

        visited.add(tuple(map(tuple, state)))

        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and tuple(map(tuple, new_state)) not in visited:
                h = manhattan_distance(new_state)
                new_g = g + 1
                new_f = new_g + h
                heapq.heappush(priority_queue, (new_f, new_state, path + [direction], new_g))

    return None

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

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()
    print("Initial State:")
    print_state(initial_state)
    print("Solving using A* search:")
    a_star_solution = a_star(initial_state)

    if a_star_solution:
        print("A* Solution:", a_star_solution)
    else:
        print("No solution found with A*.")

    print("Priority Queue (f, state, path, g):")
    for item in priority_queue:
        print(item)
```

## OUTPUT - G(N): DEPTH OF THE NODE, H(N): MANHATTAN DISTANCE

```
Enter the initial state of the 8-puzzle (0 for empty space):
Enter row 1 (space-separated): 2 8 3
Enter row 2 (space-separated): 1 6 4
Enter row 3 (space-separated): 0 7 5
Initial State:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Solving using A* search:
Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 5, 0]

Exploring state in A*:
[2, 8, 3]
[0, 6, 4]
[1, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
```

```
Exploring state in A*:
[2, 8, 3]
[1, 5, 6]
[7, 4, 0]

Exploring state in A*:
[0, 8, 3]
[2, 6, 4]
[1, 7, 5]

Exploring state in A*:
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]

Exploring state in A*:
[1, 2, 3]
[7, 8, 6]
[0, 5, 4]

Exploring state in A*:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

A* Solution: ['right', 'up', 'up', 'left', 'down', 'right']
```

# LABORATORY PROGRAM – 5(B)

## IMPLEMENT HILL CLIMBING ALGORITHM

### PSEUDOCODE OR ALGORITHM

22.10.24

LABORATORY PROGRAM - 5  
IMPLEMENT HILL CLIMBING ALGORITHM

use PSEUDOCODE for N-QUEEN PROBLEMS:

```
function HCLClimbing (initial_state, max_iterations)
    current_state = initial_state
    current_cost = CalculateCost (current_state)

    for iteration from 0 to max_iterations-1
        if current_cost == 0
            return current_state

        neighbors = GetNeighbors (current_state)
        best_neighbors = None
        best_cost = Infinity.

        for each neighbor in neighbors
            neighbor_cost = CalculateCost (neighbor)
            if neighbor_cost < best_cost
                best_cost = neighbor_cost
                best_neighbor = neighbor

        if best_cost > current_cost
            print "Local maximum reached at iteration", iteration, ". Restarting..."
            return None

        current_state = best_neighbor
        current_cost = best_cost
        print "Iteration", iteration, ": Current state:",
```

DATE: PAGE: 18/10/24

, current\_state!, cost;, current\_cost  
point "Max iterations reached without finding a solution."  
return None

## STATE SPACE TREE

ns STATE SPACE TREE FOR N-QUEEN PROBLEM :

- for 4 - Queen Problem :- state transition

(state transition) + (cost function) = final function

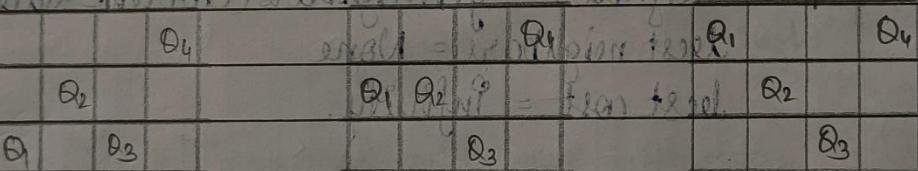
Q<sub>4</sub> Heuristic = 4.

+ initial value Q<sub>2</sub> initial cost = 4 cost

O = f(Q<sub>2</sub>) function if

at iteration 0.

(state transition) + (cost function) = final function



function f(Q<sub>i</sub>) = cost function of

(cost function) + cost(Q<sub>i</sub>) Cost = 4 + 0 Cost = 4

function > function if

function = function

local maximum reached at iteration 0.

{ No solution, only searching, as h ≤ cost  
f(Q<sub>i</sub>) + cost(Q<sub>i</sub>) ≥ f(Q<sub>i</sub>) + cost(Q<sub>i</sub>) }  
function > function if

function to handle minimum back tracking

-> function if not possible

local minimum

function f(Q<sub>i</sub>) = state function

function = function if

"state function", "not possible", "minimum" if

## CODE

```
import random

def calculate_cost(state):
    """Calculate the number of conflicts in the current state."""
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost

def get_neighbors(state):
    """Generate all possible neighbors by moving each queen in its column."""
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row: # Move the queen in column `col` to a different row
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def hill_climbing(initial_state, max_iterations=1000):
    """Perform hill climbing search to solve the N-Queens problem."""
    current_state = initial_state
    current_cost = calculate_cost(current_state)

    for iteration in range(max_iterations):
        if current_cost == 0:
            return current_state

        neighbors = get_neighbors(current_state)
        neighbor_costs = [(neighbor, calculate_cost(neighbor)) for neighbor in neighbors]
        next_state, next_cost = min(neighbor_costs, key=lambda x: x[1])

        if next_cost >= current_cost:
            print(f"Local maximum reached at iteration {iteration}. Restarting...")
            return None
        current_state, current_cost = next_state, next_cost
        print(f"Iteration {iteration}: Current state: {current_state}, Cost: {current_cost}")

    print(f"Max iterations reached without finding a solution.")
    return None

try:
    n = int(input("Enter the number of queens (N): "))
    if n <= 0:
        raise ValueError("N must be a positive integer.")

    initial_state = list(map(int, input(f"Enter the initial state as a list of {n} integers (rows for each column): ").split())))
    if len(initial_state) != n or any(not (0 <= row < n) for row in initial_state):
        raise ValueError(f"Invalid initial state. Please provide {n} integers between 0 and {n-1}.")
except ValueError as e:
    print(e)
    n = 4
    initial_state = [random.randint(0, n - 1) for _ in range(n)]
    print(f"Using random initial state: {initial_state}")

solution = None

while solution is None:
    solution = hill_climbing(initial_state)
    print(f"Solution found: {solution}")
```

## OUTPUT

```
Enter the number of queens (N): 4
Enter the initial state as a list of 4 integers (rows for each column): 3 1 2 0
Local maximum reached at iteration 0. Restarting...
```

```
Enter the number of queens (N): 4
Enter the initial state as a list of 4 integers (rows for each column): 0 0 0 0
Iteration 0: Current state: [0, 3, 0, 0], Cost: 3
Iteration 1: Current state: [1, 3, 0, 0], Cost: 1
Iteration 2: Current state: [1, 3, 0, 2], Cost: 0
Solution found: [1, 3, 0, 2]
```

