

## INDEX

Name Dipesh Sah

**Standard 5 Section 'B' Roll No. 092**

**Subject** Artificial Intelligence Lab.

## Lab - 1

### 1) Vacuum Cleaner.

Algorithm :

#### 1. Initialize goal state:

- Set goal-state = { 'A': '0'; 'B': '0' }  
(0 = clean, 1 = dirty)
- Initialize cost = 0.

#### 2. Input the vacuum location:

- Get vacuum's location:  
location\_input = 'A' or 'B'.
- Get status of the current room:  
status\_input = '0' or '1'.
- Get status of the other room:  
status\_input\_complement = '0' or '1'.

#### 3. Vacuum in Room A:

- If location\_input == 'A':
  - If status\_input == '1' (A is dirty):
    - Clean A, set goal-state['A'] = '0', increment cost.
  - If status\_input\_complement == '1'  
(B is dirty):
    - Move to B, clean it.
    - update goal-state['B'] = '0'
    - increment cost.

#### 4. Vacuum in Room B:

- If location\_input == 'B':
  - If status\_input == '1' (B is dirty):

- Clean B, set goal-state ['B'] = '0',  
• increment cost.
- If status-input-complement == '1'  
(A is dirty):
  - Move to A, clean it
  - update goal-state ['A'] = '0'.
  - increment cost.

### 5. Output :

- print goal-state and cost as the performance measure.

Output: Initial condition: { 'A': '1', 'B': '1' }

Vacuum is placed in location A.  
Location A is dirty.

Cost for cleaning A: 1

Location A has been cleaned.

Location B is Dirty.

Moving right to location B.

Cost for moving right: 2

Cost for cleaning: 3

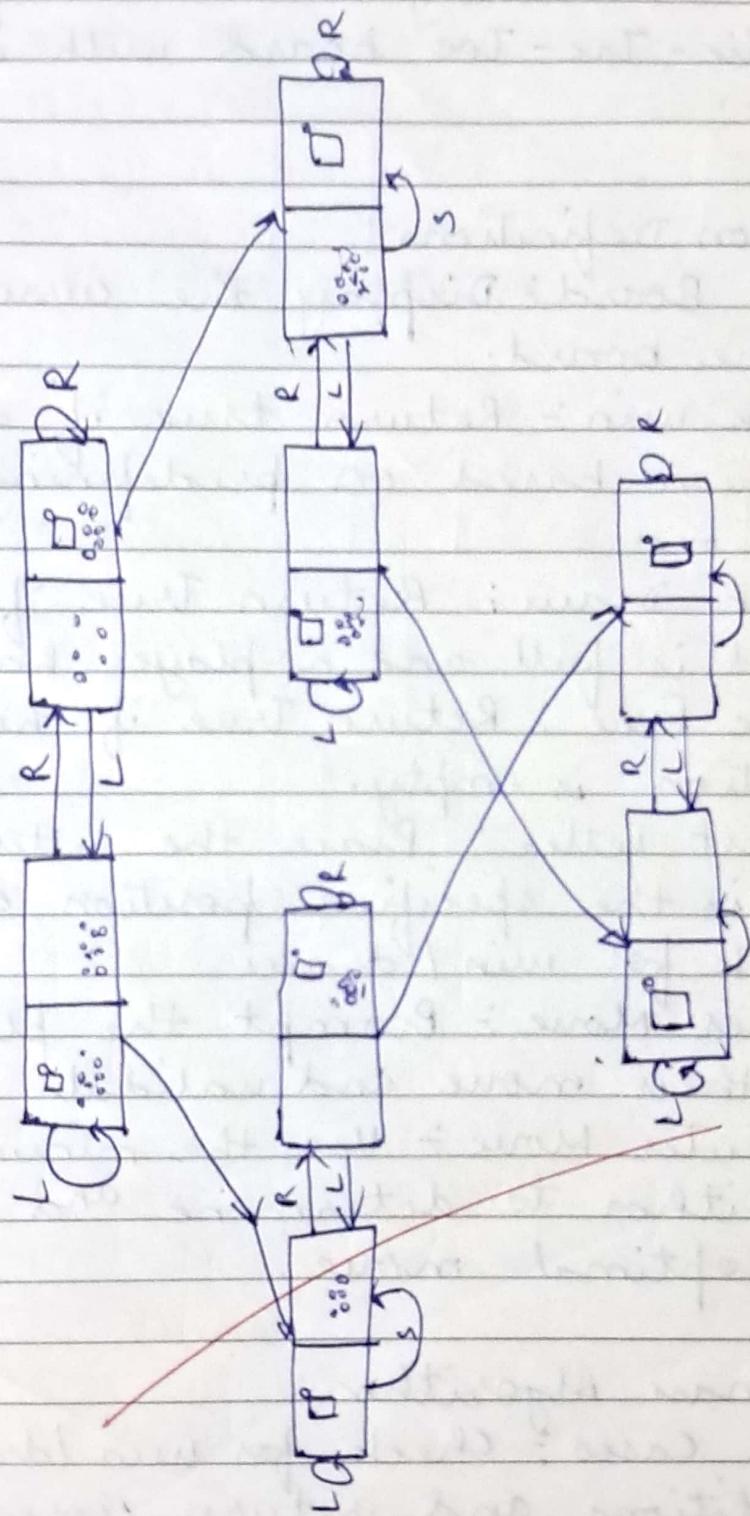
Location B has been cleaned.

GOAL State:

{ 'A': '0', 'B': '0' }

Performance Measurement: 3

## # State space tree.



## 2) Tic-Tac-Toe:

Algorithm:

### 1. Initialize the board.

Create a dictionary (or list) to represent the Tic-Tac-Toe board with empty spaces.

### 2. Function Definitions:

- Print Board: Display the current state of the board.
- check win: Return true if any player has won based on predefining winning conditions.
- check draw: Return True if the board is full and no player has won.
- space free: Return True if the selected position is empty.
- Insert letter: Place the letter ('X' or 'O') in the specified position and check for win / draw.
- Player Move: Prompt the player for their move and validate input.
- Computer Move: Use the minimax algorithm to determine and make the optimal move.

### 3. Minimax Algorithm:

- Base Cases: Check for win / draw conditions and return scores based on the outcome.
- Recursion: For maximizing (computer) and minimizing (player), evaluate

possible moves and return the best score.

#### 4. Main Game Loop.

- While the game is ongoing:
  - If no win/draw, call computerMove.
  - check for win/draw conditions.
  - If no win/draw, call player move.
  - check for win/draw conditions.

#### 5. End Game:

- Display the final board and announce the result (win/draw).

#### # Pseudocode :-

Initialize board

while true:

If not checkWin() & not checkDraw();  
compMove()

If checkWin() or checkDraw();  
break;

If not checkWin() & not checkDraw();  
playerMove()

If checkWin() or checkDraw();  
break

Display final board.

Announce result.

Output: X

Enter position for O (1-9) : 3

X	O	X	O
		X	

Enter position for O (1-9) : 7.

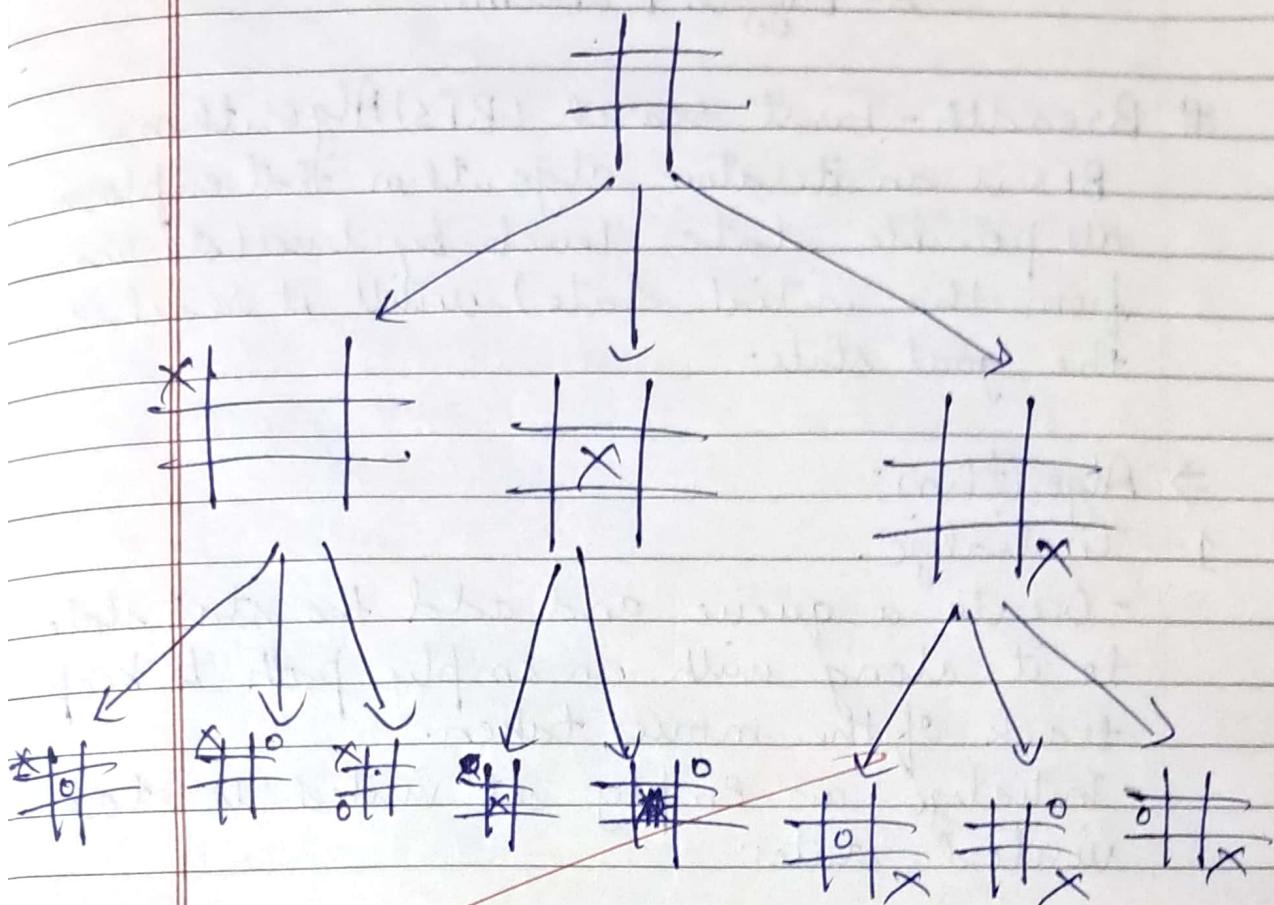
X	O	X	O
X		X	X
O		O	X

Enter position for O (1-9) : 6

X	O	X	O
X	X	X	O
O		O	X

Bot wins!

- State Space Tree:



1/10/24

## Lab - 2

### 8 - Puzzle Problem.

#### # Breadth - First search (BFS) Algorithm.

BFS is an iterative algorithm that explores all possible states level by level (starting from the initial state) until it reaches the goal state.

→ Algorithm:-

##### 1. Initialize.

- Create a queue and add the start state to it, along with an empty path to keep track of the moves taken.
- Initialize an empty set 'visited' to store visited states.

##### 2. While queue is not empty:

- dequeue the front element from the queue (current state and the path).

##### 3. Check goal.

- If the current state is the goal state, return the path (solution).

##### 4. Mark current state as visited.

- add the current state to 'visited set'.

##### 5. Generate successors:

- For each valid move from the current state:
  - apply the move to get a new state.
  - If the new state has not been visited:

- add the new state and the updated path to the queue.

#### 6. Return Failure:

- If the queue is empty and no solution has been found, return failure (no solution).

#### → Output:

Initial state:

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

Solving using BFS:

Exploring state in BFS:

$$[1, 2, 3]$$

$$[4, 0, 6]$$

$$[7, 5, 8],$$

Exploring state in BFS:

$$[1, 0, 3]$$

$$[4, 2, 6]$$

$$[7, 5, 8]$$

↑ ↑ ↑  
↓ ↓ ↓

• BFS solution: [‘down’, ‘right’]

Exploring state in DFS:

$$[1, 2, 3]$$

$$[4, 8, 6]$$

$$[7, 8, 0]$$

# 8-puzzle problem using Depth-First Search Algorithm  
DFS explores a branch as far as possible before backtracking when it reaches a dead end (or goal).

⇒ Algorithm:

1. Initialize:

- Create a stack and add the start state to it, along with an empty path to keep track of the moves taken.
- Initialize an empty set 'visited' to store visited states.

2. While stack is not empty:

- pop the top element from the stack (the current state and the path).

3. Check goal:

- If the current state is the goal, return the path (solution).

4. Mark current state as visited:

- add the current state to the 'visited set'.

5. Generate successors:

- For each valid move from the current state:
  - apply the move to get a new state.
  - If the new state has not been visited:
    - push the new state and the updated path to the stack.

6. Return failure:

- If the stack is empty and no solution has been found, return failure (no solution exists)

⇒ Output:

Solving using DFS:

Exploring state in DFS:

[1, 0, 3]

[4, 2, 6]

[7, 5, 8]

⋮ ⋮ ⋮

Exploring state in DFS:

[2, 3, 6]

[8, 0, 5]

[4, 1, 7]

⋮ ⋮ ⋮

⋮ ⋮ ⋮

⋮ ⋮ ⋮

Exploring state in DFS:

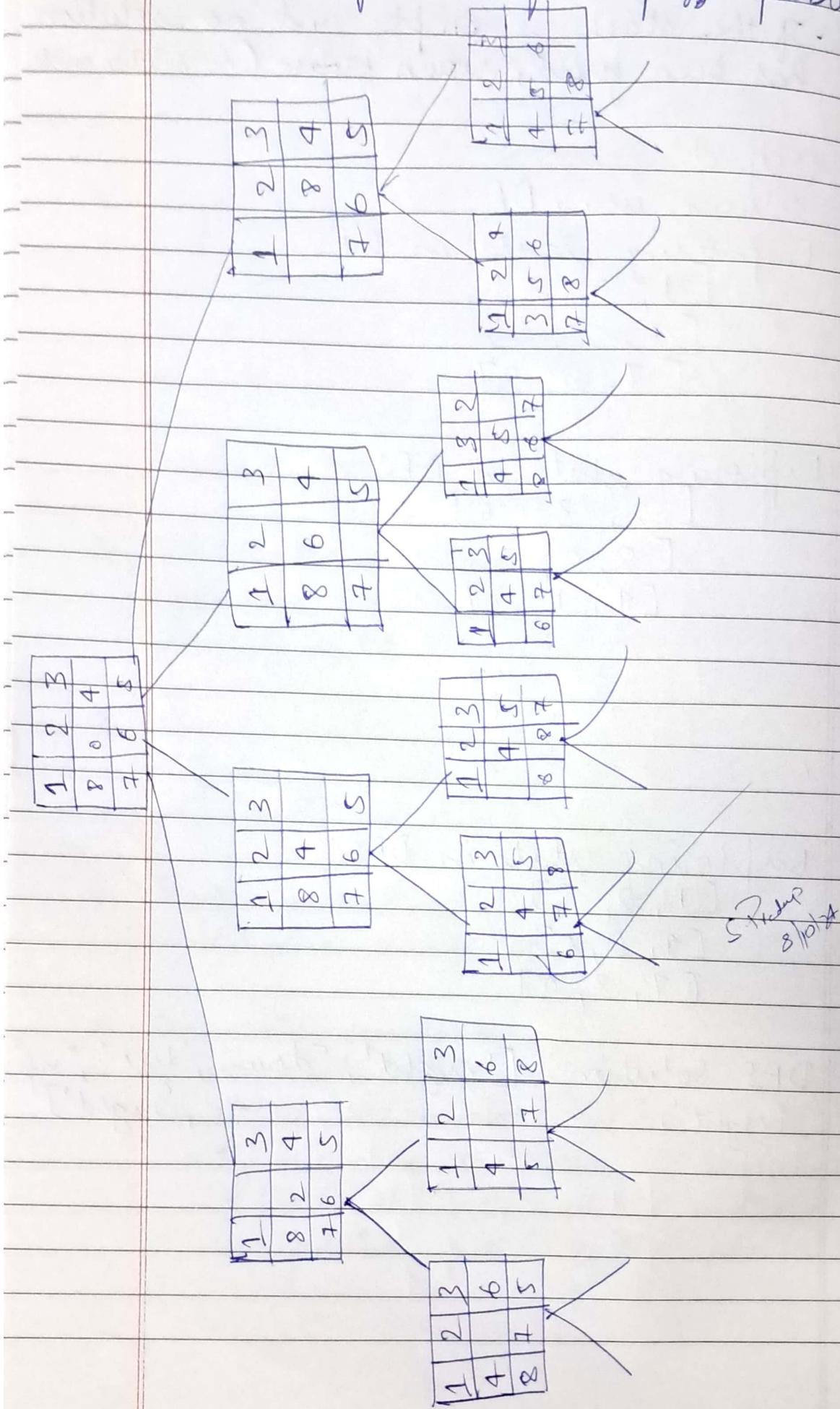
[1, 2, 3]

[7, 8, 6]

[7, 8, 0]

DFS Solution: ['right', 'down', 'left', 'up',  
'right', ..., ..., ..., ..., 'right']

## State Space Tree for 8-puzzle problem.



## Lab-3

# Algorithm 1: A\* search for 8-puzzle with Misplaced Tiles Heuristic.

### 1. Initialize

Open list  $\leftarrow \{\text{start node}\}$

Closed list  $\leftarrow \emptyset$

$g(\text{start}) \leftarrow 0$

$h(\text{start}) \leftarrow \text{number of misplaced tiles.}$

$f(\text{start}) \leftarrow g(\text{start}) + h(\text{start})$

### 2. Loop until Open list is empty

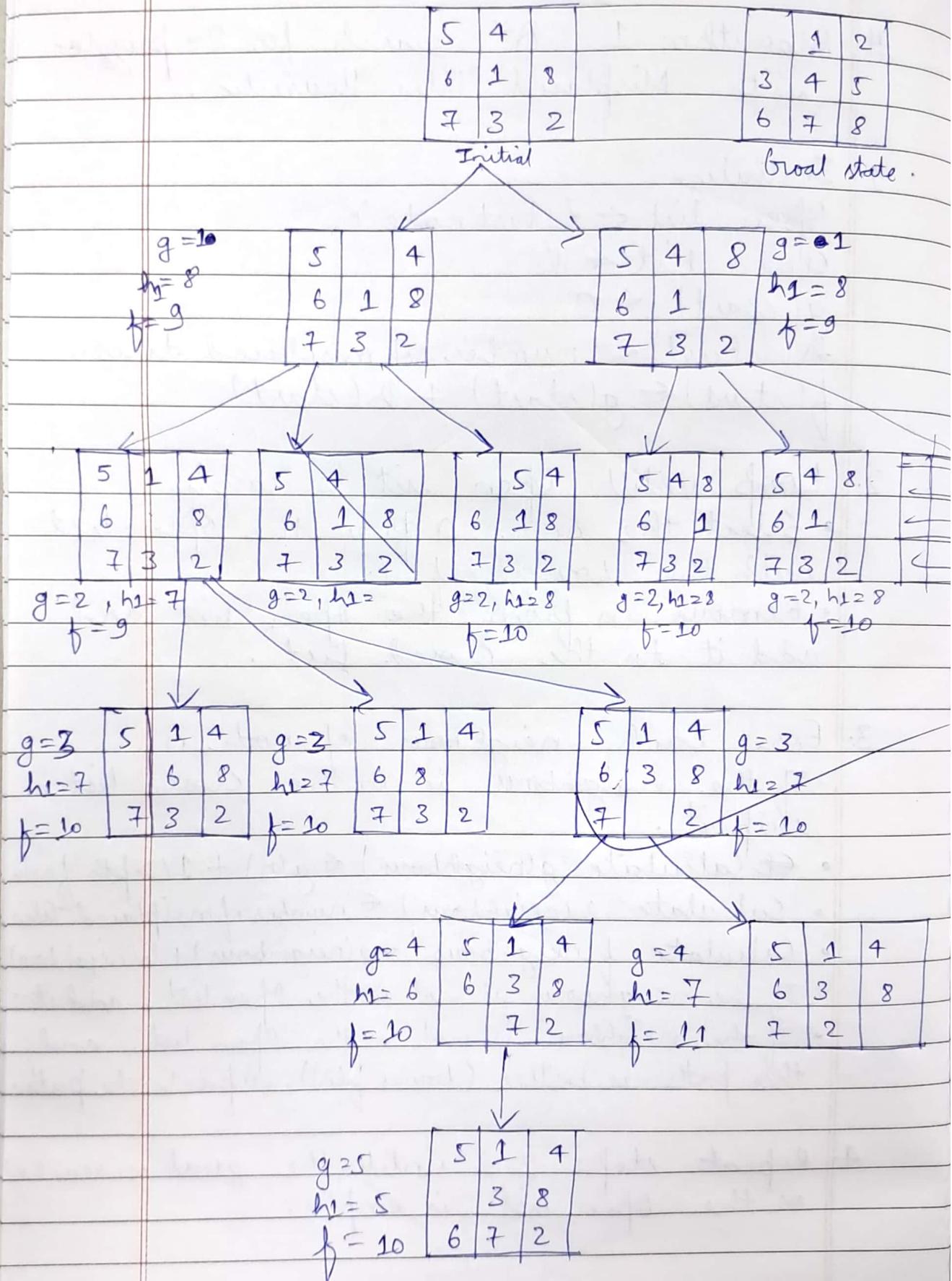
- Select the node  $n$  from the Open list with the lowest  $f(n)$ .
- Remove  $n$  from the Open list and add it to the Closed list.

### 3. For each neighbour of node $n$ .

- If the neighbour is in the Closed list, skip it.
- Calculate  $g(\text{neighbour}) \leftarrow g(n) + 1$  (depth of node)
- Calculate  $h(\text{neighbour}) \leftarrow \text{number of misplaced tiles}$
- Calculate  $f(\text{neighbour}) \leftarrow g(\text{neighbour}) + h(\text{neighbour})$
- If the neighbour is not in the Open list, add it.
- If the neighbour is not in the Open list, and this path is better (lower  $f(n)$ ), update the path.

### 4. Repeat steps 2-3 until the goal is reached or the Open list is empty.

## State Space Tree.



## # Algorithm 2: A\* Search for 8-puzzle with Manhattan Distance Heuristic.

### 1. Initialize

$\text{open list} \leftarrow \{\text{start node}\}$

$\text{closed list} \leftarrow \emptyset$

$g(\text{start}) \leftarrow 0$

$h(\text{start}) \leftarrow \text{Manhattan distance}$

$f(\text{start}) \leftarrow g(\text{start}) + h(\text{start})$

### 2. Loop until open list is empty.

- Select the node  $n$  from the open list with the lowest  $f(n)$
- If  $n$  is the goal state/node, terminate and return the path.
- Remove  $n$  from the open list and add it to the closed list.

### 3. For each neighbour of node $n$ ,

- If the neighbour is in the closed list, skip it.
- Calculate  $g(\text{neighbour}) \leftarrow g(n) + 1$  (depth of node)
- Calculate  $h(\text{neighbour}) \leftarrow \text{Manhattan Distance}$ .
- Calculate  $f(\text{neighbour}) \leftarrow g(\text{neighbour}) + h(\text{neighbour})$
- If the neighbour is not in the open list, add it.
- If the neighbour is in the open list and this path is better(lower( $f(n)$ )), update the path.

### 4. Repeat steps 2-3 until the goal is reached or the open list is empty.

## State Space Tree

Initial

$g=0$	2	8	3		1	2	3
$h_2=4$	1		4		8		4
$f=4+0=4$	7	6	5		7	6	5

$g=1$	2	8	3	$g=1$	2	3	$g(n)=1$	2	8	3
$h(n)=1+1+2+1$ =5	1	4		$h(n)=3$	1	8	$h(n)=5$	2	4	
$f=6$	7	6	5	$f(n)=4$	7	6	$f(n)=6$	7	6	5

$g(n)=2$		2	3	$g(n)=2$	2	3	
$h(n)=2+1=3$	1	8	4	$h(n)=2+1+1+1$	1	8	4
$f(n)=4$	7	6	5	$f(n)=2+4=6$	7	6	5

1	2	3	$g(n)=3$
8		4	$h(n)=1$
7	6	5	$f(n)=3+1=4$

1	2	3		1	2	3	
8		4		7	8	4	
7	6	5		6	5		

S. Rishabh  
15/10/2024

Final state  
goal reached.

## Lab - 4

# Algorithm for : Implement Hill Climbing Search algorithm to solve N-Queens problem.

1. Generate a random initial configuration of N queens on the board with one queen per column.
2. Evaluate the cost (heuristic cost) of the current configuration. This cost is the number of pairs of queens that are attacking each other.
3. Generate neighbours by moving each queen within its column to every other row and calculate the cost for each neighbour.
4. Select the neighbour with the lowest cost. If no neighbour has a lower cost than the current state, terminate.  
(reached a local or global optimum).
5. If a neighbour has a lower cost, move to that neighbour and repeat the process.
6. If stuck at a local optimum, optionally restart with a new random configuration (random restart hill climbing).
7. Stop when the goal (zero conflict) is reached.

# Pseudocode:

function hill-climbing-n-queens( initial state )  
current-state = initial-state

while true:

    current\_heuristic = calculate\_heuristic( current )

    if current\_heuristic == 0:

        return current\_state

    neighbours = generate\_neighbours( current\_state )

    best\_neighbour = None

    best\_heuristic = infinity

    for each neighbour in neighbours:

        heuristic = calculate\_heuristic(neighbour)

        if heuristic < best\_heuristic

            best\_heuristic = heuristic

            best\_neighbour = neighbour

    if best\_heuristic >= current\_heuristic:

        break // No better neighbours found.

    current\_state = best\_neighbour

return None // No solutions found.

## State Space Tree

Step. 0:

				$\theta$
		$\theta$	$\theta$	$\theta$
initial state	$\theta$		$\theta$	

$n_0 = 3, n_1 = 1, n_2 = 2, n_3 = 0$   
cost = 2

Step. 1:- Move to neighbours

				$\theta$
	$\theta$			$\theta$
			$\theta$	
	$\theta$			

$n_0 = 3, n_1 = 0$   
 $n_2 = 2, n_3 = 1$   
cost = 1

Step. 2: Move to better neighbours.

				$\theta$
	$\theta$			$\theta$
			$\theta$	
	$\theta$			

$n_0 = 2, n_1 = 0$   
 $n_2 = 3, n_3 = 1$   
cost = 0

→ Final state (Optimal)

Step. 3: Reached local minimum already,  
so no better solution can be found.

				$\theta$
	$\theta$			$\theta$
			$\theta$	
	$\theta$			

$n_0 = 2, n_1 = 0$   
 $n_2 = 2, n_3 = 1$   
Final cost = 0

S.Pradeep  
22/10/24