INDEX

Name Dipesh Sah.

Standard 5 Section 'B' Roll No. 092

Subject Artificial Intelligence Lab.

SL No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	01-Oct 2024	Vacuum cleaner; Tic-Tac-Toe.	01 - 07	8/10
02.	8-Oct 2024	8-puzzle problem-DFS & BFS	8 - 12	S.P. 8/10/24

Lab - 1

1) Vaccum Cleaner.

Algorithm:

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

2. Input the vaccum location:
   - Get vaccum'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. Vaccum 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. Vaccum 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.

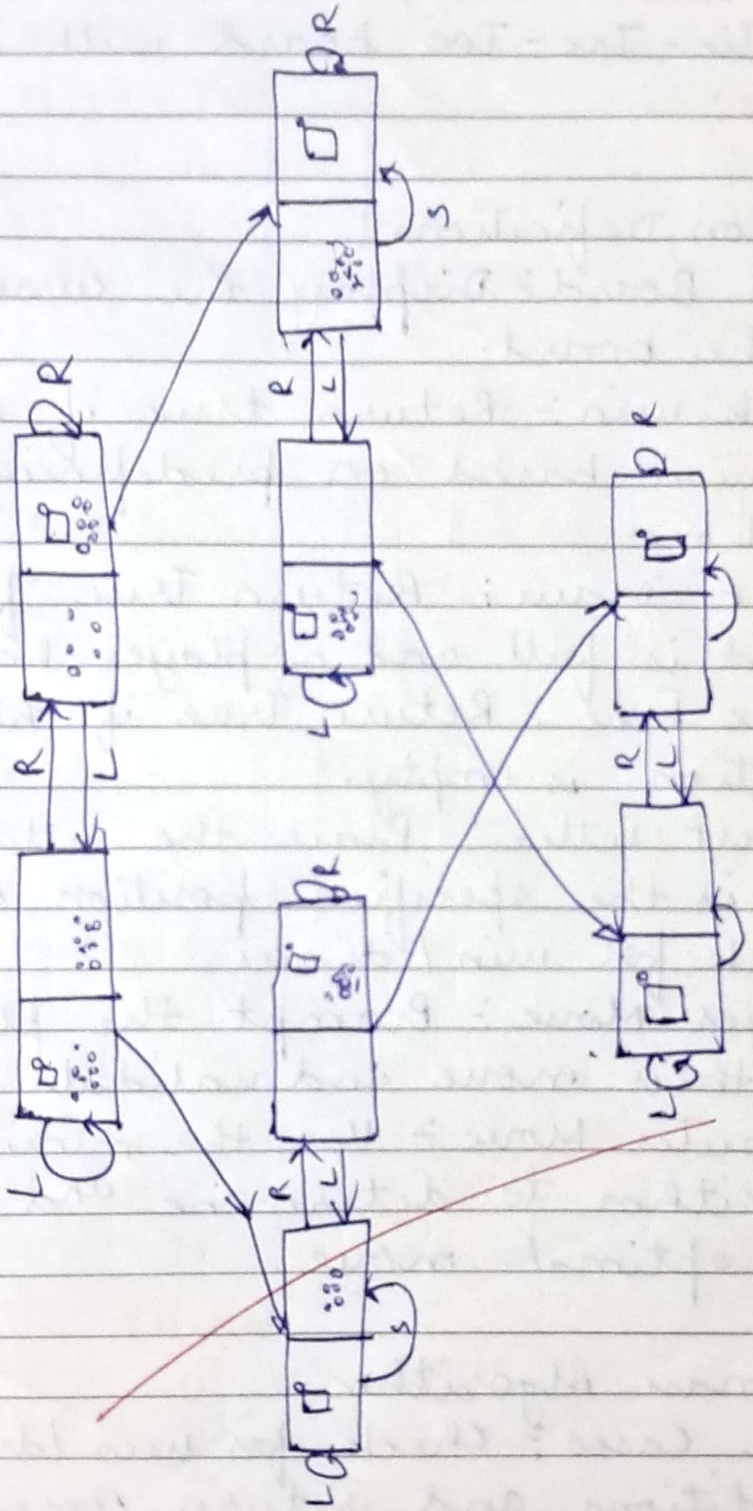
5. Output:
   - print goal_state and cost as the performance measure.

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 wining 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.

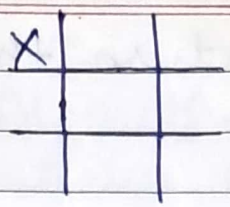
4. Main Game Loop.
- While the game is ongoing:
  - If no win/draw, call computer Move.
  - Check for win/draw conditions.
  - If no win/draw, call Player move.
  - Check for win/draw conditions.

5. End Game:
- Display the final board and announce the result (win/draw).

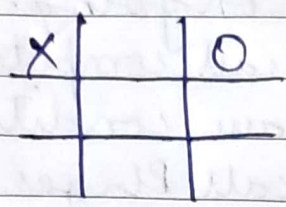
Pseudocode :-

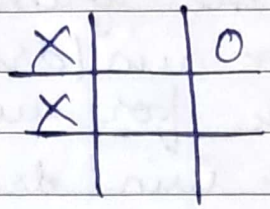
```
Initialize board
While true:
    If not checkWin() && not checkDraw();
        compMove()
    If checkWin() or checkDraw();
        break;
    If not checkWin() && not checkDraw();
        playerMove()
    If checkWin() or checkDraw():
        break

Display final board.
Announce result.
```

Output:

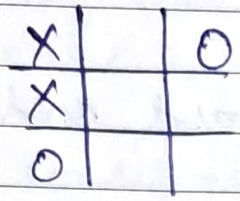


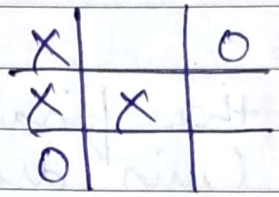
Enter position for O (1-9) : 3



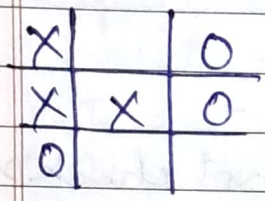


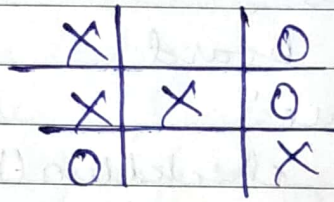
Enter position for O (1-9) : 7.





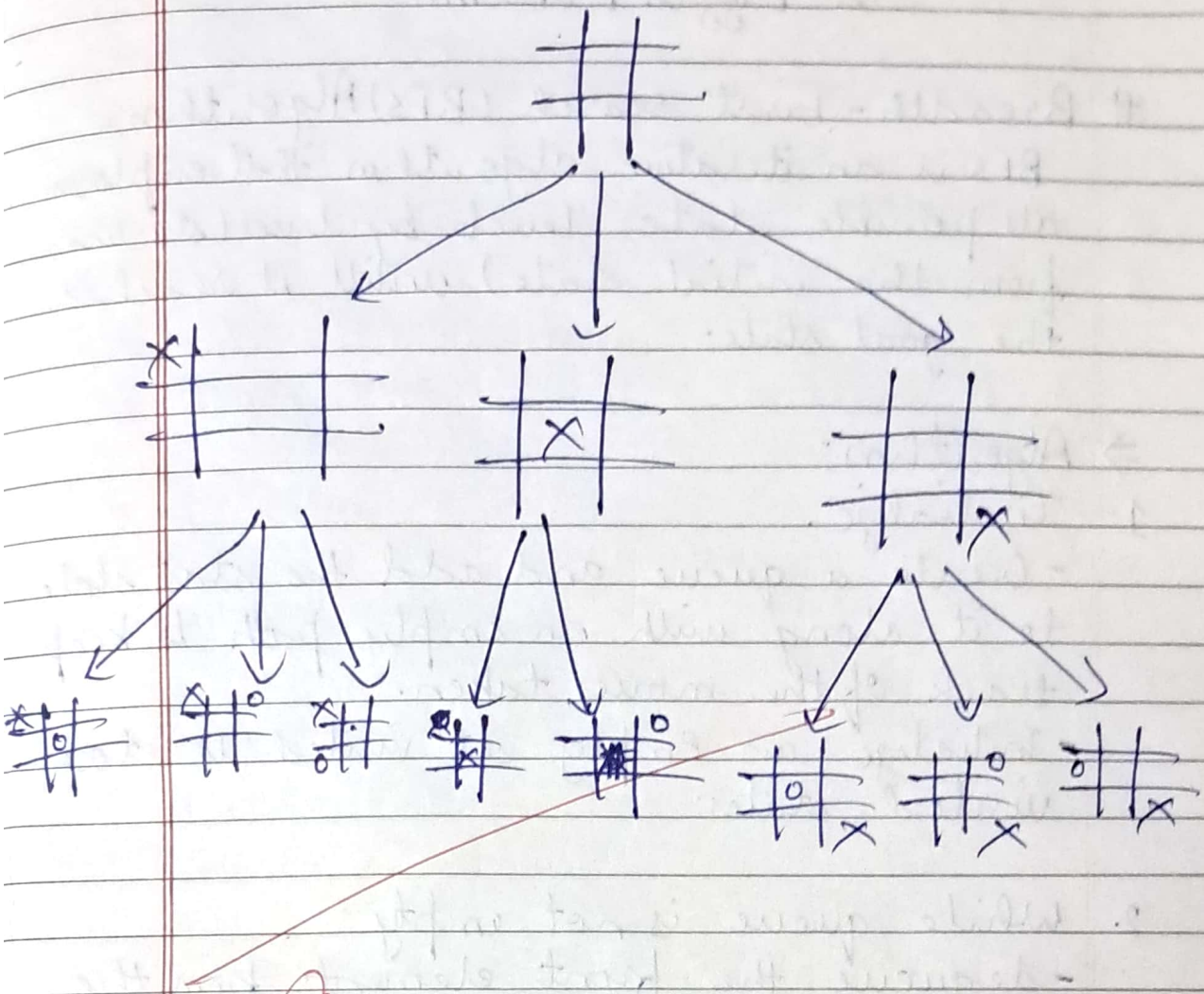
Enter position for O (1-9) : 6





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:

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

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, 5, 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.

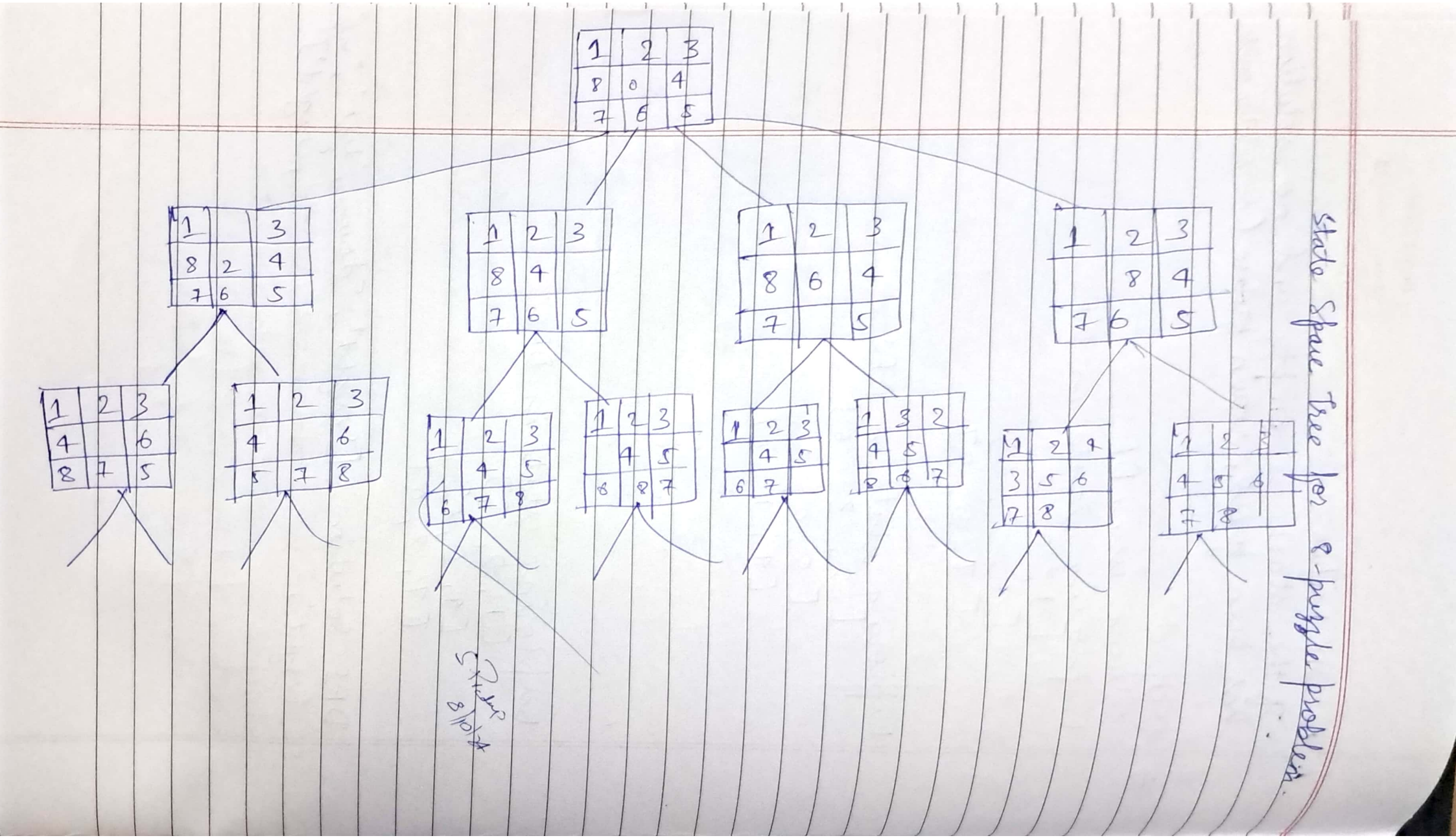
6. Return failure:
- If the stack is empty and no solution has been found, return failure (no solution exists).

⇒ 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]
[4, 5, 6]
[7, 8, 0]

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


State Space Tree for 8-puzzle problem.