LAB - 1

Implement a vaccum cleaner agent

Algorithm:-

1) Initialize

set goal_state = {'A' : '0', 'B' : '0'}
set cost = 0

2) Input

- Get the current location of vacuum cleaner as 'A' or 'B'
- Get the status of the current location, where '0' means clean and '1' means dirty (status_input)
- Get the status of the other location, where '0' means clean and '1' means dirty (status_input_complement)
- If the vaccum cleaner is at location A:
  If location A is dirty (status_input == '1')
  Clean location A (set goal_state['A'] = '0')
  Increament the cost by 1 for cleaning location A
  Print the status of cleaning
- If the vaccum cleaner is at location B:
  - If location B is dirty (status_inpt_complemen == '1')
  - Move to location B (increment the cost by 1 for movement)
  - Clean location B (set goal_state ['B'] = '0')
  - Increament the cost by 1 for cleaning location B
  - If location B is clean (status_input_complement == '0'):
  - No action needed for location B

If the vaccum cleaner is at location B:

If location B is dirty (status_input == '1'):

- Clean location B (set goal state ['B'] = '0').
- Incrcament the cost by 1 for cleaning location B
- Print the status of cleaning

If location A is dirty (status_input_complement == '1'):

- Move to location A (increment the cost by 1 for movement).
- Clean location A (set goal_state ['A'] = '0').
- Increment the cost by 1 for cleaning location A
- If location A is clean (status_input_complement == '0'):
- No action needed for location A

Expected output :-

- Print the final goal_state (both locations should be clean.
- Print the total cost (cleaning & moving actions

Output :-

Enter Location of Vaccum (A or B) : A

Enter Status of A (0 for Clean, 1 for Dirty) : 1

Enter status of other room (0 for clean, 1 for Dirty) : 0

Initial location Condition : {'A' : '0', 'B' : '0'}

Vaccum is placed in location A

Location A is Dirty

Cost for Cleaning A : 1

Location A has been cleanes

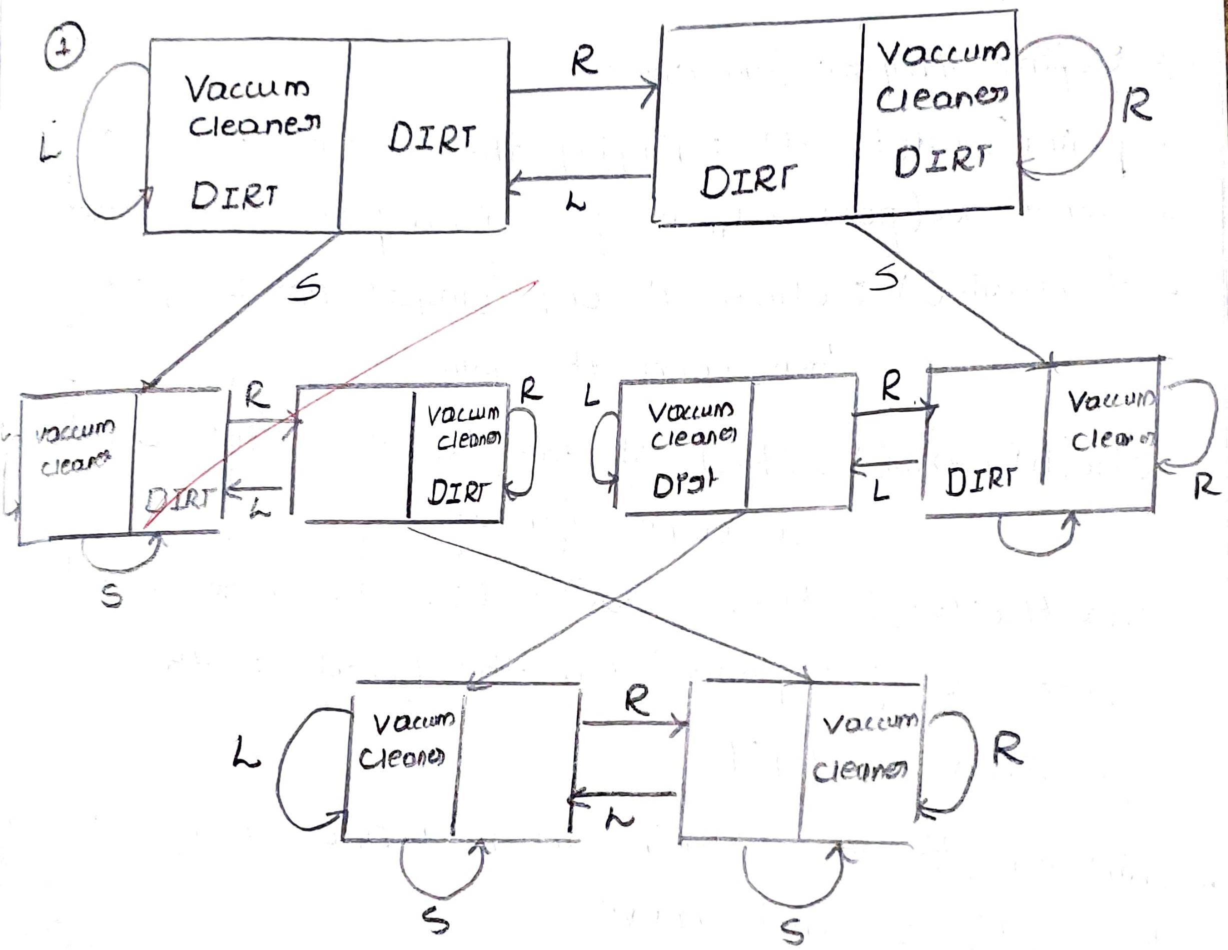
Location B is already clean. No action. Total cost

GOAL STATE:

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

Performance Measurement (Total cost) : 1

STATE SPACE TREE



2) TIC-TAC TOE

Algorithm

1) Initialize Game Board:

- Create an empty board with positions labeled 1-9

2) Define Helper Functions:

- printBoard(board) : Display the current board state.
- spaceFree(pos): Check if a given position is empty
- checkWin() : Check if any player (bot or player) has won the game.
- checkDraw() : Check if the board is full without a winner (draw
- insertLetter(letter, position) : Place a letter ('X' for bot, 'O' for player) on the board if the position is empty.
- minimax (board, isMaximizing): Minimax algorithm for bot's optimal move

3) Main Game Loop:

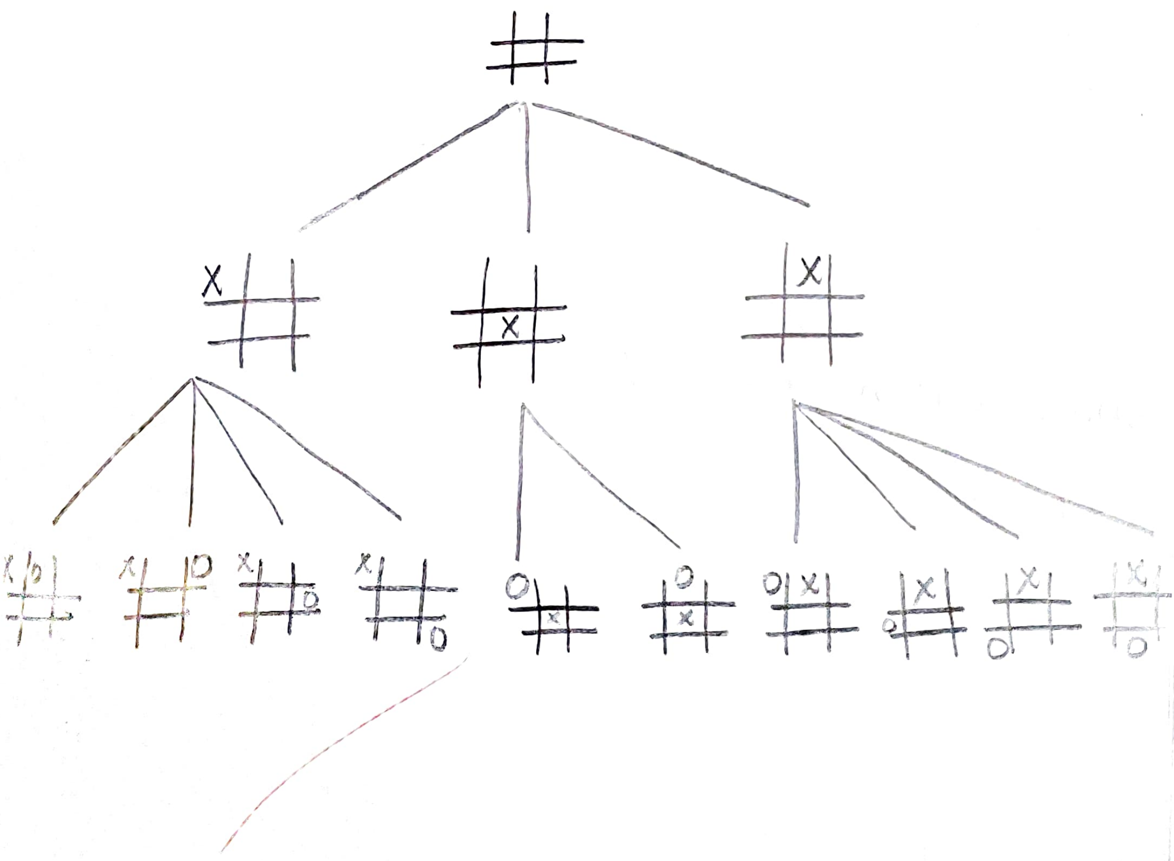
- Players Turn:
  - Prompt the player to input a valid position (1-9)
  - Check if the move results in a win or draw. If so, end the game.

- Bot's Turn:
  - * Use minimax() to calculate the best move for the bot.
  - * Make the move on the board & check if the bot wins or if the game results in a draw.

4) Game End Conditions:

- If a win or draw condition is met, display the result (either 'You win' 'Bot Win or 'draw')
- Exit the game loop when the game ends

STATE SPACE TREE



Output:

X		

Enter position for O : 1
Position taken, please pick a different position.
Enter new position : 3

X		O

X		O
X		

Enter position for O : 7

X		O
X		
O		

X		O
X	X	
O		

Enter position for O : 6

X		O
X	X	O
O		

X		O
X	X	O
O		X

Bot wins!

25/8/25