

## 1. Implement Tic –Tac –Toe Game.

**Code:**

```
board={1:'',2:'',3:'',
       4:'',5:'',6:'',
       7:'',8:'',9:''
}

print("Sonal , 1BM22CS286")

def printBoard(board):
    print(board[1]+'|'+board[2]+'|'+board[3])
    print('---')
    print(board[4] + '|' + board[5] + '|' + board[6])
    print('---')
    print(board[7] + '|' + board[8] + '|' + board[9])
    print('\n')

def spaceFree(pos):
    if(board[pos]==' '):
        return True
    else:
        return False

def checkWin():
    if(board[1]==board[2] and board[1]==board[3] and board[1]!=' '):
        return True
    elif(board[4]==board[5] and board[4]==board[6] and board[4]!=' '):
        return True
    elif(board[7]==board[8] and board[7]==board[9] and board[7]!=' '):
        return True
    elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):
        return True
    elif (board[3] == board[5] and board[3] == board[7] and board[3] != ' '):
        return True
    elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):
        return True
    elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):
        return True
    elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):
        return True
    else:
        return False
```

```

def checkMoveForWin(move):
    if (board[1]==board[2] and board[1]==board[3] and board[1] ==move):
        return True
    elif (board[4]==board[5] and board[4]==board[6] and board[4] ==move):
        return True
    elif (board[7]==board[8] and board[7]==board[9] and board[7] ==move):
        return True
    elif (board[1]==board[5] and board[1]==board[9] and board[1] ==move):
        return True
    elif (board[3]==board[5] and board[3]==board[7] and board[3] ==move):
        return True
    elif (board[1]==board[4] and board[1]==board[7] and board[1] ==move):
        return True
    elif (board[2]==board[5] and board[2]==board[8] and board[2] ==move):
        return True
    elif (board[3]==board[6] and board[3]==board[9] and board[3] ==move):
        return True
    else:
        return False

def checkDraw():
    for key in board.keys():
        if (board[key]==' '):
            return False
    return True


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)

```

```
return

player = 'O'
bot ='X'

def playerMove():
    position=int(input('Enter position for O:'))
    insertLetter(player, position)
    return

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)
    return

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] = ''
                if (score > bestScore):
                    bestScore = score

        return bestScore
```

```

else:
    bestScore = 1000

for key in board.keys():
    if board[key] == '':
        board[key] = player
        score = minimax(board, True)
        board[key] = ''
    if (score < bestScore):
        bestScore = score
return bestScore

while not checkWin():
    playerMove()
    compMove()

```

## Output :

```

Sonal , 1BM22CS286
Enter position for 0:1
0| |
-+-
| |
-+-
| |

Enter position for 0:9
0|X|0
-+-
0|X|X
-+-
X|0|0
-+-
| |

Draw!

```

## Observation book screenshots:

1.10.24

Date \_\_\_\_\_ / \_\_\_\_\_  
Page \_\_\_\_\_

Tic Tac Toe

```
function minimax (node, depth, isMaximizingPlayer)
if node is a terminal state
    return evaluate (node)
set up with maxDepth = depth, target = target
if isMaximizingPlayer:
    bestValue = -infinity
    for each child node in
        value = minimax (child, depth+1, false)
        bestValue = max (bestValue, value)
    return bestValue
else:
    bestValue = +infinity
    for each child node in
        value = minimax (child, depth+1, true)
        bestValue = min (bestValue, value)
    return bestValue
```

uhol24

## 2. Implement vacuum cleaner agent.

**Code:**

```
def vacuum_world():
    # Initializing goal_state
    # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    print("Sonal,1BM22CS286")

    location_input = input("Enter Location of Vacuum (A or B): ").strip().upper() # User input of location
    status_input = input("Enter status of {location_input} (0 for Clean, 1 for Dirty): ").strip()
    status_input_complement = input("Enter status of other room (0 for Clean, 1 for Dirty): ").strip()

    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' # Clean A
            cost += 1 # Cost for sucking
            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 # Cost for moving right
            print("Cost for moving RIGHT: " + str(cost))

            goal_state['B'] = '0' # Clean B
            cost += 1 # Cost for sucking
            print("Cost for SUCK: " + str(cost))
            print("Location B has been Cleaned.")

        else:
            print("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 # Cost for moving right
print("Cost for moving RIGHT: " + str(cost))

goal_state['B'] = '0' # Clean B
cost += 1 # Cost for sucking
print("Cost for SUCK: " + str(cost))
print("Location B has been Cleaned.")
else:
    print("Location B is already clean.")

elif location_input == 'B':
    print("Vacuum is placed in Location B")
    if status_input == '1':
        print("Location B is Dirty.")
        goal_state['B'] = '0' # Clean B
        cost += 1 # Cost for sucking
        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 # Cost for moving left
        print("Cost for moving LEFT: " + str(cost))

        goal_state['A'] = '0' # Clean A
        cost += 1 # Cost for sucking
        print("Cost for SUCK: " + str(cost))
        print("Location A has been Cleaned.")
    else:
        print("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 # Cost for moving left
        print("Cost for moving LEFT: " + str(cost))

        goal_state['A'] = '0' # Clean A
        cost += 1 # Cost for sucking
        print("Cost for SUCK: " + str(cost))
        print("Location A has been Cleaned.")
    else:
        print("Location A is already clean.")

```

```
# Done cleaning
print("GOAL STATE: ")
print(goal_state)
print("Performance Measurement: " + str(cost))

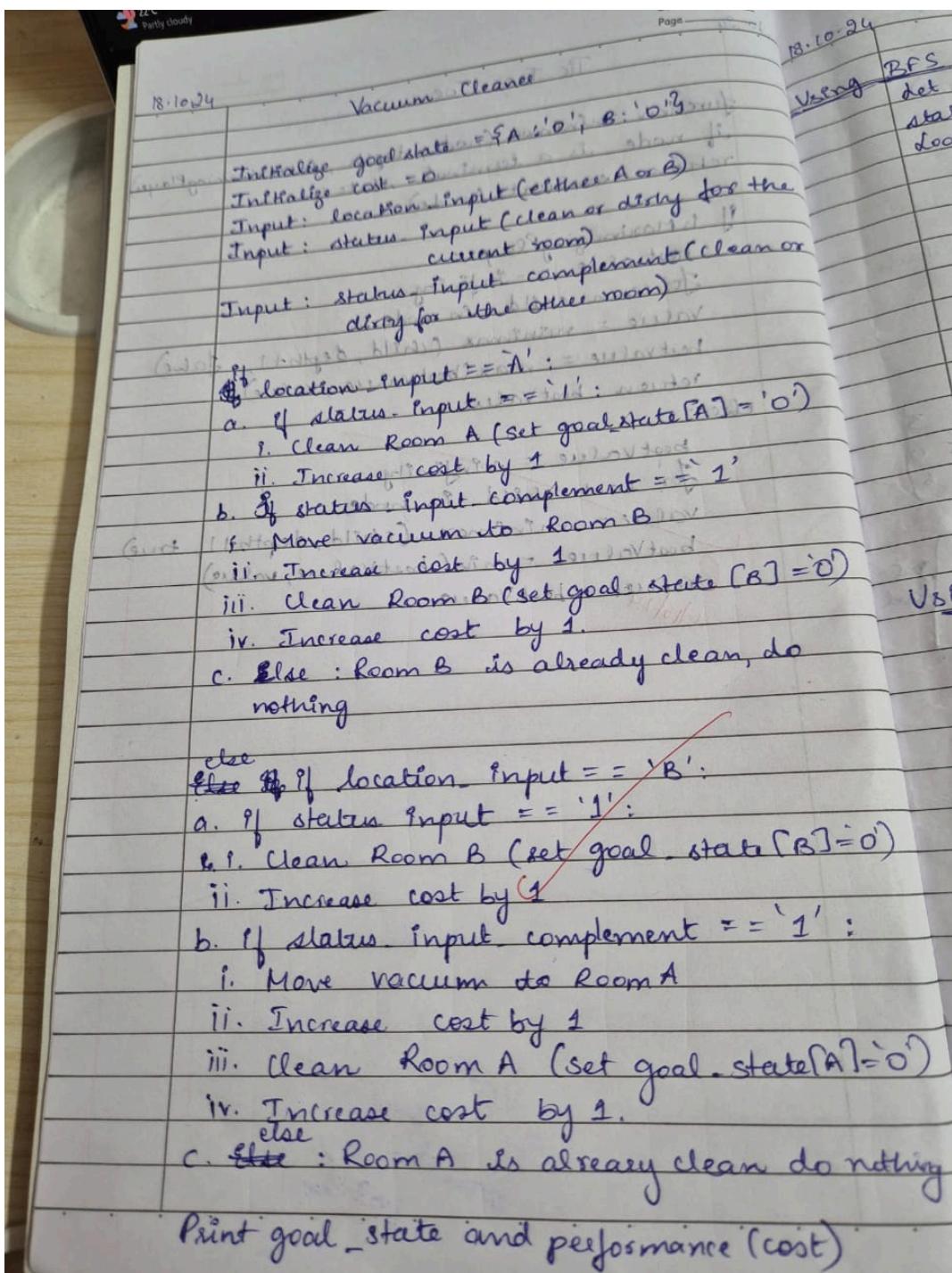
# Output
vacuum_world()
```

## Output:

---

→ Sonal,1BM22CS286  
Enter Location of Vacuum (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): 1  
Initial Location Condition: {'A': '0', 'B': '0'}  
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 SUCK: 3  
Location B has been Cleaned.  
GOAL STATE:  
{'A': '0', 'B': '0'}  
Performance Measurement: 3

## Observation book screenshots:



### 3. Solve 8 puzzle problem using BFS and DFS.

#### USING BFS

Code:

```
from collections import deque
print("Sonal,1BM22CS286")

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[new_row * 3 + col], new_board[new_row * 3 + new_col] = new_board[new_row * 3 + new_col], new_board[new_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()

```

**Output :**

```
Sonal,1BM22CS286
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'):
1 2 3 4 0 6 7 5 8
Current Board State:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Current Board State:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

Current Board State:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

Current Board State:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 6, 8]
[7, 5, 0]

Current Board State:
[1, 2, 0]
[4, 6, 3]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Solution found in 2 steps.
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

## Observation book screenshots:

Using BFS

let fringe be a list containing the initial state.

loop

the

or

if fringe is empty return failure

Node  $\leftarrow$  remove - first (fringe)

if Node is a goal

then return the path from initial state to Node

else

general all successors of Node and add generated node to the back of the fringe

End loop.

## 8 Puzzle

State space tree BFS

Initial state [1 2 3] a level of (initial)  
[4 0 5]  
[9 8 6] (move 0 up)

[1 2 3] [1 2 3] [1 2 3] move down  
[0 4 5] [4 5 0] [4 0 5]  
[3 8 6] [7 8 6] [7 8 6] move right

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

[1 2 3] [1 2 3] [1 2 3] [1 2 3] goal  
[4 5 6] [4 5 0] [4 0 5] [4 5 6] level  
[A 8 0] [7 8 6] [2 8 6] [2 8 0]

## USING DFS

**Code :**

```
from collections import deque
print("Sonal,1BM22CS286")

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 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 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.")

```

**Output:**

```

Streaming output truncated to the last 5000 lines.
[2, 5, 6]
[0, 3, 1]
[4, 7, 8]

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

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

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

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

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

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

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

```

**Observation book screenshots:**

Using DFS

Let fringe be a list containing the initial state.

Loop

if fringe is empty return failure.

Node  $\leftarrow$  remove first (fringe)

if Node is a goal

then return the path from initial state to Node

else

general all successors of Node and  
add generated node to the ~~fringe~~ <sup>front</sup> of  
~~the for DFS of fringe.~~

End loop.

DFS

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

for

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

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

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

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

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

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

#### 4. Solve 8 puzzle problem using A\* algorithm

##### Using misplaced tiles

Code:

```
import heapq
print("Sonal,1BM22CS286")

# Define the goal state for the 8-puzzle
GOAL_STATE = [
    [2, 8, 1],
    [0, 4, 3],
    [7, 6, 5]
]

# Define the position moves (up, down, left, right)
MOVES = [
    (-1, 0), # Up
    (1, 0), # Down
    (0, -1), # Left
    (0, 1) # Right
]

class PuzzleNode:
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g # Cost from start to current node
        self.h = h # Heuristic cost to goal
        self.f = g + h # Total cost

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

def misplaced_tiles(state):
    """Heuristic function that counts the number of misplaced tiles."""
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != GOAL_STATE[i][j]:
                misplaced += 1
    return misplaced
```

```

def get_zero_position(state):
    """Find the position of the zero (empty tile) in the puzzle."""
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

def generate_successors(node):
    """Generate successors by moving the empty tile in all possible directions."""
    successors = []
    zero_x, zero_y = get_zero_position(node.state)

    for move_x, move_y in MOVES:
        new_x, new_y = zero_x + move_x, zero_y + move_y
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in node.state]
            new_state[zero_x][zero_y], new_state[new_x][new_y] = new_state[new_x][new_y],
            new_state[zero_x][zero_y]
            h = misplaced_tiles(new_state)
            successors.append(PuzzleNode(new_state, parent=node, g=node.g + 1, h=h))
    return successors

def is_goal(state):
    """Check if the current state is the goal state."""
    return state == GOAL_STATE

def reconstruct_path(node):
    """Reconstruct the path from the start state to the goal state."""
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]

def a_star(start_state):
    """A* algorithm to solve the 8-puzzle problem."""
    start_node = PuzzleNode(start_state, g=0, h=misplaced_tiles(start_state))
    open_list = []
    closed_set = set()

    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

```

```

if is_goal(current_node.state):
    return reconstruct_path(current_node)

closed_set.add(tuple(map(tuple, current_node.state)))

for successor in generate_successors(current_node):
    if tuple(map(tuple, successor.state)) in closed_set:
        continue
    heapq.heappush(open_list, successor)

return None

def get_user_input():
    """Get a valid 8-puzzle input state from the user."""
    print("Enter your 8-puzzle configuration (0 represents the empty tile):")
    state = []
    values = set()

    for i in range(3):
        row = input(f"Enter row {i+1} (space-separated numbers between 0 and 8): ").split()
        if len(row) != 3:
            print("Each row must have exactly 3 numbers. Please try again.")
            return None

        row = [int(x) for x in row]

        if not all(0 <= x <= 8 for x in row):
            print("Values must be between 0 and 8. Please try again.")
            return None

        state.append(row)
        values.update(row)

    if values != set(range(9)):
        print("All numbers from 0 to 8 must be present exactly once. Please try again.")
        return None

    return state

# Main function
def main():
    start_state = None
    while start_state is None:
        start_state = get_user_input()

```

```

solution = a_star(start_state)

# Print the solution steps
if solution:
    print("Solution found in", len(solution) - 1, "moves:")
    for step in solution:
        for row in step:
            print(row)
            print()
else:
    print("No solution found.")

if __name__ == "__main__":
    main()

```

## Output:

```

Sonal,1BM22CS286
Enter your 8-puzzle configuration (0 represents the empty tile):
Enter row 1 (space-separated numbers between 0 and 8): 1 2 3
Enter row 2 (space-separated numbers between 0 and 8): 8 0 4
Enter row 3 (space-separated numbers between 0 and 8): 7 6 5
Solution found in 9 moves:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

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

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

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

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

```

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

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

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

### Observation book screenshots:

A\* Algorithm

function A\* search (problem) returns a solution or failure

node  $\leftarrow$  a node  $n$  with  $n$ .state = problem.

initialState,  $n.g = 0$

frontier  $\leftarrow$  a priority queue ordered by ascending  $g(n)$ , only element  $n$

loop do

if empty? (frontier) then return failure

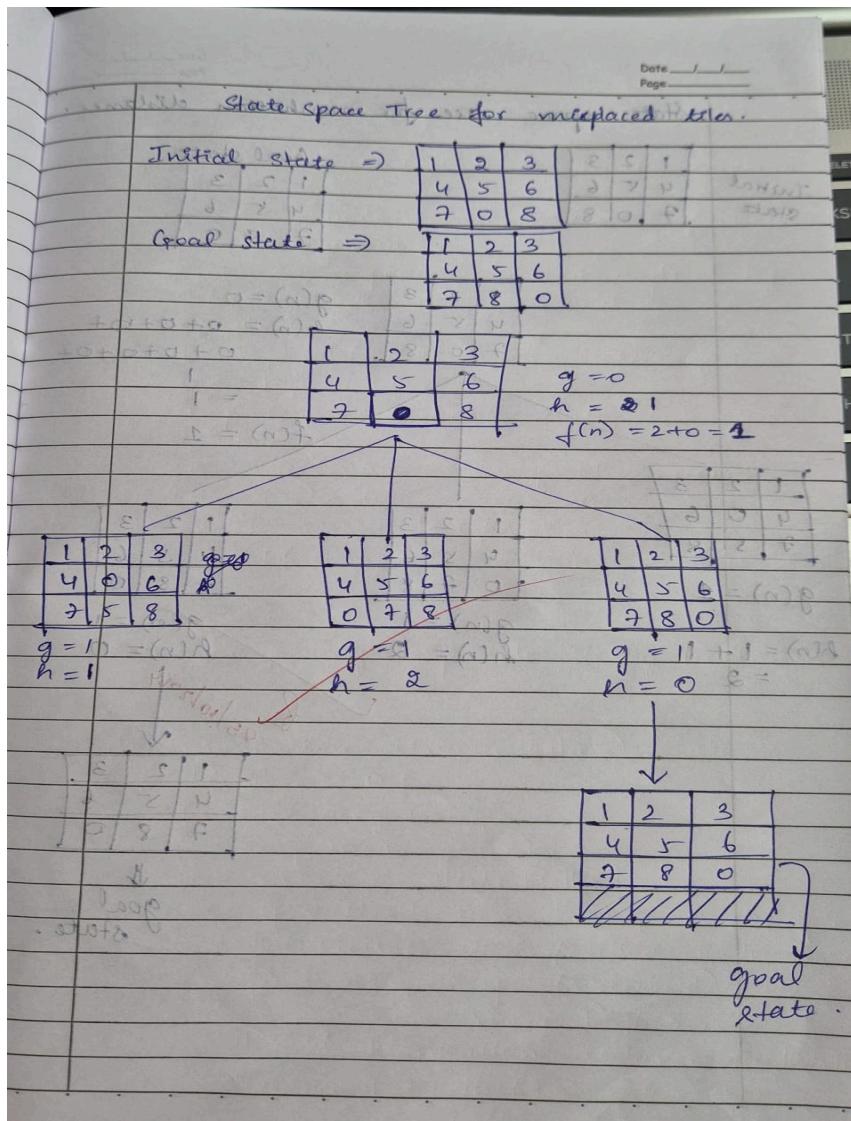
$n \leftarrow \text{pop}(\text{frontier})$

if problem.goalTest ( $n$ .state) then return solution( $n$ )

for each action  $a$  in problem.action ( $n$ .state) do

$n' \leftarrow \text{childNode}(\text{problem}, n, a)$

~~insert ( $n'.g(n') + h(n')$ , frontier)~~



## Using Manhattan distance:

**Code:**

```
# manhattan distance
import heapq
print("SonaliBM22CS286")

# Define the goal state for the 8-puzzle
GOAL_STATE = [
    [2, 8, 1],
    [0, 4, 3],
```

```
[7, 6, 5]
```

```
]
```

```
# Define the position moves (up, down, left, right)
MOVES = [
    (-1, 0), # Up
    (1, 0), # Down
    (0, -1), # Left
    (0, 1) # Right
]

class PuzzleNode:
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g # Cost from start to current node
        self.h = h # Heuristic cost to goal (Manhattan distance)
        self.f = g + h # Total cost

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

def manhattan_distance(state):
    """Heuristic function that calculates the Manhattan distance for each tile."""
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0: # Skip the empty tile
                # Calculate goal position for this value
                goal_x, goal_y = (value - 1) // 3, (value - 1) % 3
                # Add the Manhattan distance for this tile
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

def get_zero_position(state):
    """Find the position of the zero (empty tile) in the puzzle."""
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

def generate_successors(node):
```

```

"""Generate successors by moving the empty tile in all possible directions."""
successors = []
zero_x, zero_y = get_zero_position(node.state)

for move_x, move_y in MOVES:
    new_x, new_y = zero_x + move_x, zero_y + move_y
    if 0 <= new_x < 3 and 0 <= new_y < 3:
        new_state = [row[:] for row in node.state]
        new_state[zero_x][zero_y], new_state[new_x][new_y] = new_state[new_x][new_y],
        new_state[zero_x][zero_y]
        h = manhattan_distance(new_state)
        successors.append(PuzzleNode(new_state, parent=node, g=node.g + 1, h=h))
return successors

def is_goal(state):
    """Check if the current state is the goal state."""
    return state == GOAL_STATE

def reconstruct_path(node):
    """Reconstruct the path from the start state to the goal state."""
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]

def a_star(start_state):
    """A* algorithm to solve the 8-puzzle problem."""
    start_node = PuzzleNode(start_state, g=0, h=manhattan_distance(start_state))
    open_list = []
    closed_set = set()

    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if is_goal(current_node.state):
            return reconstruct_path(current_node)

        closed_set.add(tuple(map(tuple, current_node.state)))

        for successor in generate_successors(current_node):
            if tuple(map(tuple, successor.state)) in closed_set:
                continue

```

```

    heapq.heappush(open_list, successor)

    return None

def get_user_input():
    """Get a valid 8-puzzle input state from the user."""
    print("Enter your 8-puzzle configuration (0 represents the empty tile):")
    state = []
    values = set()

    for i in range(3):
        row = input(f"Enter row {i+1} (space-separated numbers between 0 and 8): ").split()
        if len(row) != 3:
            print("Each row must have exactly 3 numbers. Please try again.")
            return None

        row = [int(x) for x in row]

        if not all(0 <= x <= 8 for x in row):
            print("Values must be between 0 and 8. Please try again.")
            return None

        state.append(row)
        values.update(row)

    if values != set(range(9)):
        print("All numbers from 0 to 8 must be present exactly once. Please try again.")
        return None

    return state

# Main function
def main():
    start_state = None
    while start_state is None:
        start_state = get_user_input()

    solution = a_star(start_state)

    # Print the solution steps
    if solution:
        print("Solution found in", len(solution) - 1, "moves:")
        for step in solution:
            for row in step:
                print(row)

```

```

        print()
else:
    print("No solution found.")

if __name__ == "__main__":
    main()

```

## Output:

---

```

Sonal,1BM22CS286
Enter your 8-puzzle configuration (0 represents the empty tile):
Enter row 1 (space-separated numbers between 0 and 8): 1 2 3
Enter row 2 (space-separated numbers between 0 and 8): 8 0 4
Enter row 3 (space-separated numbers between 0 and 8): 7 6 5
Solution found in 9 moves:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

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

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

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

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

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

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

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

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

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

```

## **Observation book screenshots:**

Logic for solving 8 puzzle using Manhattan distance.

State space tree for manhattan distance

Initial state

1	2	3
4	5	6
7	0	8

Goal state

1	2	3
4	5	6
7	8	0

1	2	3
4	5	6
7	0	8

$$g(n) = 0$$

$$h(n) = 0 + 0 + 0 + \\ 0 + 0 + 0 + 0 +$$

$$= \frac{1}{1}$$

$$f(n) = 1$$

1	2	3
4	0	6
7	5	8

$$g(n) = 1$$

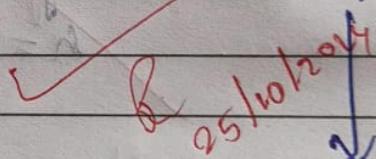
$$\begin{aligned} h(n) &= 1 + 1 \\ &= 2 \end{aligned}$$

1	2	3
4	5	6
0	7	8

$$\begin{aligned}g(n) &= 1 \\h(n) &= 2\end{aligned}$$

1	2	3
4	5	6
7	8	0

$$\begin{aligned}g(n) &= 1 \\h(n) &= 0\end{aligned}$$



1	2	3
4	5	6
7	8	0

goal state