1. Implement Tic-Tac-Toe Game

Code:

```
#Implementing tic-tac toe game
import math

# Board is represented as a list of 9 elements, initially empty
board = [' ' for _ in range(9)]

# Function to print the board
def print_board(board):
    for row in [board[i * 3:(i + 1) * 3] for i in range(3)]:
        print('| ' + ' | '.join(row) + ' |')

# Function to check if there is a winner
def winner(board, player):
    win_conditions = [(0, 1, 2), (3, 4, 5), (6, 7, 8), (0, 3, 6), (1, 4, 7), (2, 5, 8), (0, 4, 8), (2, 4, 6)]
    for condition in win_conditions:
        if all([board[i] == player for i in condition]):
            return True
    return False

# Function to check if the board is full
def is_board_full(board):
    return ' ' not in board

# Function to make a move
def make_move(board, move, player):
    board[move] = player

# Minimax function to find the best move
def minimax(board, depth, is_maximizing):
    if winner(board, 'O'):  # AI wins
        return 1
    if winner(board, 'X'):  # Player wins
        return-1
    if is_board_full(board):  # Draw
        return 0
    if is_maximizing:
        best_score =-math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'O'
                score = minimax(board, depth + 1, False)
                board[i] = ' '
                best_score = max(score, best_score)
        return best_score
    else:
        best_score = math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'X'
                score = minimax(board, depth + 1, True)
                board[i] = ' '
                best_score = min(score, best_score)
        return best_score
```

```
# Function to find the best move for the AI
def find_best_move(board):
    best_move = None
    best_score =-math.inf
    for i in range(9):
        if board[i] == ' ':
            board[i] = 'O'
            score = minimax(board, 0, False)
            board[i] = ' '
            if score > best_score:
                best_score = score
                best_move = i
    return best_move

# Main game loop
def play_game():
    while True:
        print_board(board)
        # Player's move
        move = int(input("Enter your move (0-8): "))
        if board[move] != ' ':
            print("Invalid move. Try again.")
            continue
        make_move(board, move, 'X')

        if winner(board, 'X'):
            print_board(board)
            print("Player wins!")
            break
        if is_board_full(board):
            print_board(board)
            print("It's a draw!")
            break
        # AI's move
        print("AI is making a move...")
        ai_move = find_best_move(board)
        make_move(board, ai_move, 'O')

        if winner(board, 'O'):
            print_board(board)
            print("AI wins!")
            break
        if is_board_full(board):
            print_board(board)
            print("It's a draw!")
            break
play_game()
```

Output:

```
|   |   |   |
|   |   |   |
|   |   |   |
Enter your move (0-8): 1
AI is making a move...
| O | X |   |
|   |   |   |
```

| | | |
Enter your move (0-8): 4
AI is making a move...
O	X	
	X	
	O	
Enter your move (0-8): 2
AI is making a move...
O	X	X
	X	
O	O	
Enter your move (0-8): 5
AI is making a move...
O	X	X
O	X	X
O	O	
AI wins!

Pseudo Code:

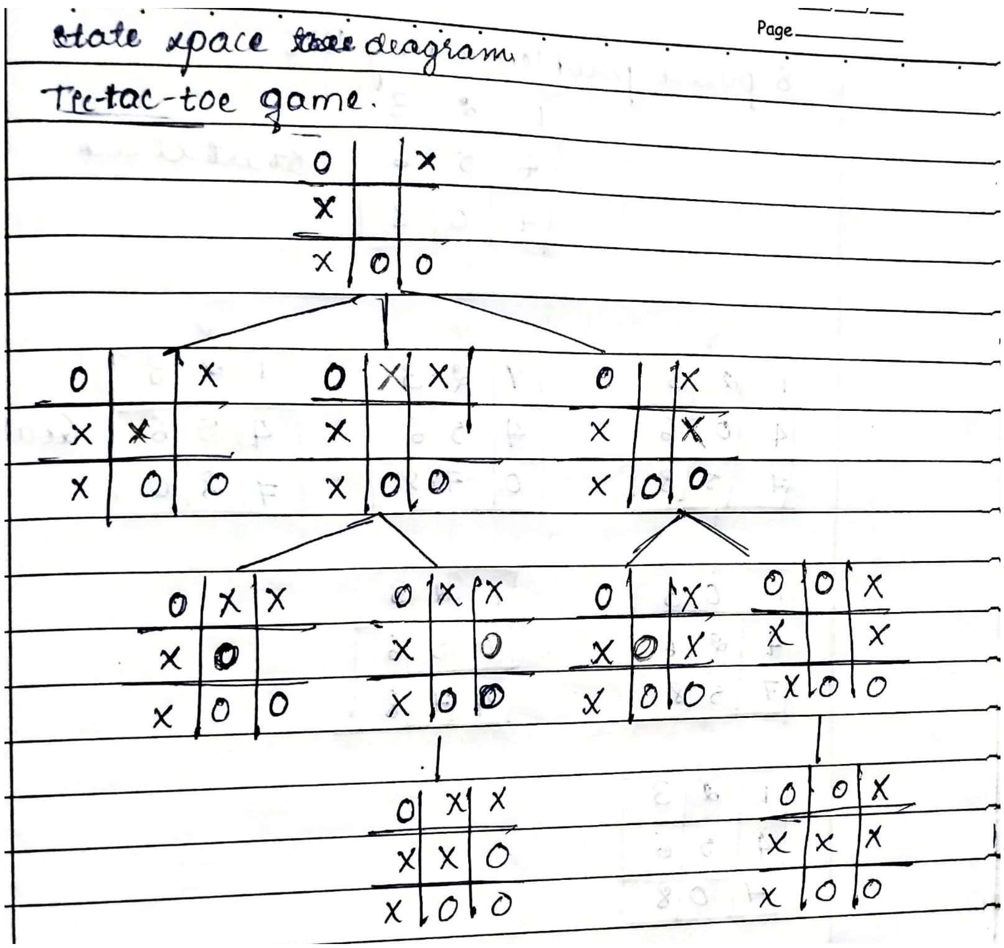
1. Implementing tic-tac-toe game.

Algorithm:

1. Initialize the 3X3 board using lists in which cells are empty initially.
2. ~~Take the~~ input from the user to choose the ~~position~~.
3. If the position is non empty return invalid input.
4. ~~Else~~

3) Write a function to display board ~~to~~ and current state of the board.
4) Write a function to check for win, check all rows, column and diagonal wins.
5) Write a function to check for draw
6) Define a function to accept the ~~human~~ input for human player's move.
If the cell is empty, then add 'X' to the cell.
7) The main game function, display the current board get human player's move, check for win or draw, else get bot's move, check for win or draw.
8) For AI moves, define a function to find the best move so that AI wins. ~~Check~~ Finding a better move by keeping 'O' in every empty cell.

State Space Diagram:



2. Solve 8-Puzzle Problem (DFS)

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

# Example initial state (you can modify this)
initial_state = [[1, 2, 3],
                 [4, 5, 6],
                 [7, 0, 8]]

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:

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


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


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


DFS Solution: ['right']
Time taken by DFS: 0.000517 seconds
```

Pseudo Code:

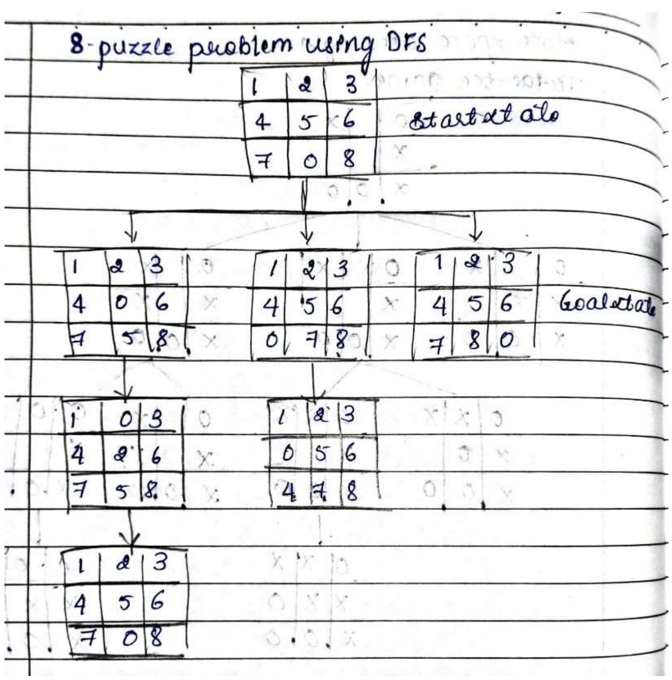
Solving 8 puzzle problems:

Pseudocode:

```
Class node:
    Function_init (state, parent=none, action=none,
    path-cost=0):
        SET self.state = state
        SET self.parent = parent
        SET self.action = action
        SET self.path_cost = path-cost
    Function expand():
        CREATE children
        SET row, col = find_blank()
        CREATE possible_actions
        IF row > 0 THEN ADD 'Up' to possible-actions
        IF row < 2 THEN ADD 'Down' to possibleaction
        IF col > 0 THEN ADD 'Left' to possible-action
        FOR action IN possible_actions:
            CREATE new_state as a copy of self.state
            IF action == 'Up' THEN SWAP new-state[row]
            [col] with new-state[row-1][col]
            ELSE IF action == 'Down' THEN SWAP
            new_state[row][col] with new_state[row+1]
            [col]
            ELSE IF action == 'Left' THEN SWAP new_state
            [row][col] with new-state[row][col -1]
            ELSE IF action == 'Right' THEN SWAP new_
            state[row][col] with new_state[row][col+1]
            APPEND new node(new_state, self, action,
            self.path_cost+1) to children.
        RETURN children
    FUNCTION find-blank():
        FOR row FROM 0 TO 2:
            FOR col FROM 0 to 2:
                IF self.state[row][col]==0 THEN
                RETURN row, col
FUNCTION depth-first_search(initial_state,
goal-state):
    SET frontier = [node(initial_state)]
    SET exploded = empty set
    WHILE frontier is not empty:
        SET node = frontier.pop()
        IF node.state == goal_state
    THEN RETURN node
        ADD tuple of node.state to explored
        FOR child IN node.expand():
            IF tuple of child.state NOT IN exploded
        THEN APPEND child to frontier
        RETURN none
FUNCTION print_solution(node):
    CREATE path
    WHILE node is not None:
```

```
APPEND (node.action, node.state) to path
    SET node = node.parent
    REVERSE path
    FOR (action, state) IN path:
        IF action is not none THEN
        PRINT "Action:", action
            PRINT state
            PRINT ""
SET initial_state = [[1,2,3],[0,4,6],[7,5,8]]
SET goal_state = [[1,2,3],[4,5,6],[7,8,0]]
SET solution = depth_first_search(initial_state, goal_state)
IF solution is not none THEN PRINT "Solution found:"
CALL print_solution(solution)
ELSE
    PRINT "Solution not found"
```

State Space Diagram:



3. Implementing Iterative Deepening Search Algorithm.

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 iterative_deepening_dfs(initial_state):
    def depth_limited_dfs(state, path, depth_limit):
        print("Exploring state in DLS:")
        print_state(state)

        if is_goal(state):
            return path

        if depth_limit == 0:
            return None
```

```
        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and str(new_state) not in visited:
                visited.add(str(new_state))
                result = depth_limited_dfs(new_state, path + [direction], depth_limit- 1)
                if result is not None:
                    return result
        return None

    depth_limit = 0
    while True:
        visited = set()
        result = depth_limited_dfs(initial_state, [], depth_limit)
        if result is not None:
            return result
        depth_limit += 1
        if depth_limit > 20:  # Adjust the limit based on your puzzle complexity
            return None

# Example initial state (you can modify this)
initial_state = [[1, 2, 3],
                 [4, 5, 6],
                 [7, 0, 8]]

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

start_time_iddfs = time.time()
print("Solving using Iterative Deepening DFS:")
iddfs_solution = iterative_deepening_dfs(initial_state)
end_time_iddfs = time.time()

if iddfs_solution:
    print("IDDFS Solution:", iddfs_solution)
else:
    print("No solution found with IDDFS.")
print(f"Time taken by IDDFS: {end_time_iddfs- start_time_iddfs:.6f} seconds")
```

Output:

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


Solving using Iterative Deepening DFS:
Exploring state in DLS:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
```

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

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

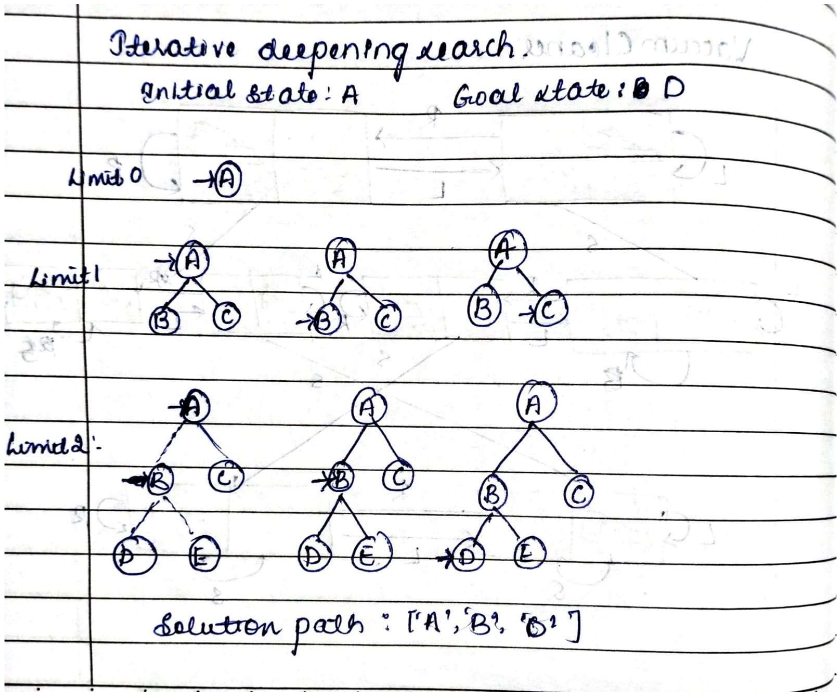
Pseudo Code:

a) Implementing Iterative Deepening Search.

Pseudocode:

```
FUNCTION iterative_deepening_search(initial_state,
goal_state, depth):
    FOR depth FROM 0 TO max-depth:
        SET result = depth_limited_search(initial_state,
        goal_state, depth)
        IF result is not none THEN
            RETURN result
    RETURN none
FUNCTION depth_limited_search(node, goal_state,
limit):
    IF node.state == goal_state THEN RETURN node
    IF node.depth >= limit THEN RETURN none
    FOR each child IN expand(node):
        SET result = depth_limited_search(child, goal_state,
        limit)
        IF result is not None THEN RETURN result
    RETURN none
SET initial_state
SET goal_state
SET max_depth
SET solution = iterative_deepening_search(initial_state,
goal_state, max_depth)
IF solution is not none THEN PRINT solution
ELSE PRINT "no solution found"
```

State Space Diagram:



4. Implement Vacuum Cleaner Agent.

Code:

```
def vacuum_cleaner_agent(location, status):
    x, y = location
    if status[x][y] == 'Dirty':
        return f"The vacuum cleaner is at ({x}, {y}) and it is dirty. Cleaning."
    else:
        return f"The vacuum cleaner is at ({x}, {y}) and it is clean. Moving."

status = [['Dirty', 'Clean'], ['Dirty', 'Dirty']]
location = (0, 0)

while True:
    action = vacuum_cleaner_agent(location, status)
    print(action)

    x, y = location
    if status[x][y] == 'Dirty':
        status[x][y] = 'Clean'

    if status[0][0] == 'Clean' and status[0][1] == 'Clean' and status[1][0] == 'Clean' and status[1][1] ==
'Clean':
        print("All locations are clean. The vacuum cleaner is finished.")
        break

    if y < 1:
        location = (x, y + 1)
    elif x < 1:
        location = (x + 1, 0)
```

Output:

The vacuum cleaner is at (0, 0) and it is dirty. Cleaning.
The vacuum cleaner is at (0, 1) and it is clean. Moving.
The vacuum cleaner is at (1, 0) and it is dirty. Cleaning.
The vacuum cleaner is at (1, 1) and it is dirty. Cleaning.
All locations are clean. The vacuum cleaner is finished.

Pseudo Code:

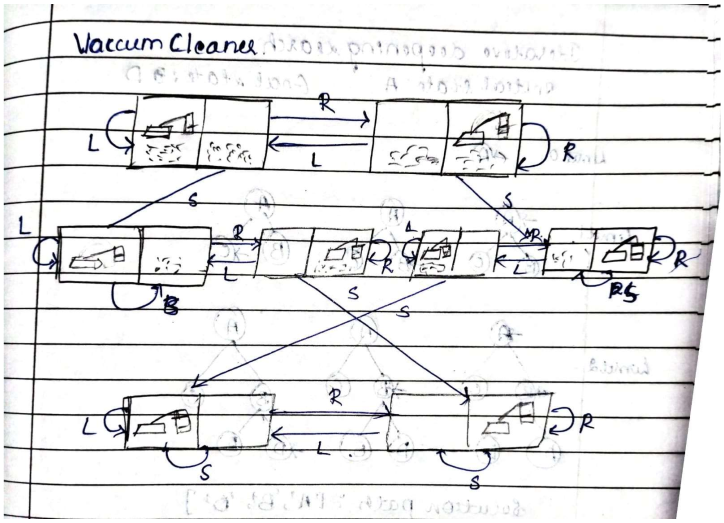
Page

4. Implementing. Vaccum cleaner agent

Algorithm.

1. Write a function to state the status of the location. [Dirty & clean].
2. Initialize the location to (0,0)
3. Till ~~the~~ all the locations are cleaned, check wheather the location is ~~cleaned~~ dirty, If yes clean, else move to the next location.
4. If all the locations are cleaned, state that the ~~locations~~ cleaning is finished.
5. Else move to the next location.

State Space Diagram:



5. Implementing A* Algorithm(Misplaced Tiles)

Code:

```
#Heuristic approach to 8-puzzle problem
import heapq

def solve_8puzzle(initial_state):
    goal_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
    priority_queue = [(heuristic(initial_state, goal_state), 0, initial_state, [])]
    visited = set()

    while priority_queue:
        f_cost, g_cost, current_state, current_path = heapq.heappop(priority_queue)

        if current_state == goal_state:
            return current_path + [current_state]

        if tuple(map(tuple, current_state)) in visited:
            continue
        visited.add(tuple(map(tuple, current_state)))

        for next_state, action in get_possible_moves(current_state):
            new_g_cost = g_cost + 1
            new_f_cost = new_g_cost + heuristic(next_state, goal_state)
            heapq.heappush(priority_queue, (new_f_cost, new_g_cost, next_state, current_path +
[(current_state, action)]))

    return None

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

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

def get_possible_moves(state):
    row, col = find_position(state, 0)
    possible_moves = []

    if row > 0:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row- 1][col] = new_state[row- 1][col], new_state[row][col]
        possible_moves.append((new_state, 'Up'))
    if row < 2:
        new_state = [list(row) for row in state]
```

```
        new_state[row][col], new_state[row + 1][col] = new_state[row + 1][col], new_state[row][col]
        possible_moves.append((new_state, 'Down'))
    if col > 0:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row][col- 1] = new_state[row][col- 1], new_state[row][col]
        possible_moves.append((new_state, 'Left'))
    if col < 2:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row][col + 1] = new_state[row][col + 1], new_state[row][col]
        possible_moves.append((new_state, 'Right'))

    return possible_moves

initial_state = [[2, 8, 3], [1, 6, 4], [0, 7, 5]]
solution = solve_8puzzle(initial_state)

if solution:
    print("Solution found:")
    for state, action in solution[:-1]:
        print("--------------------")
        for row in state:
            print(row)
        print("Move:", action)
    print("--------------------")
    for row in solution[-1]:
        print(row)
else:
    print("No solution found.")
```

Output:

```
Solution found:
--------------------
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]
Move: Right
--------------------
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Move: Up
--------------------
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Move: Up
--------------------
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Move: Left
--------------------
```

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

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

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

Pseudo Code:

Page

Pseudocode for Heuristic (misplaced tiles.)

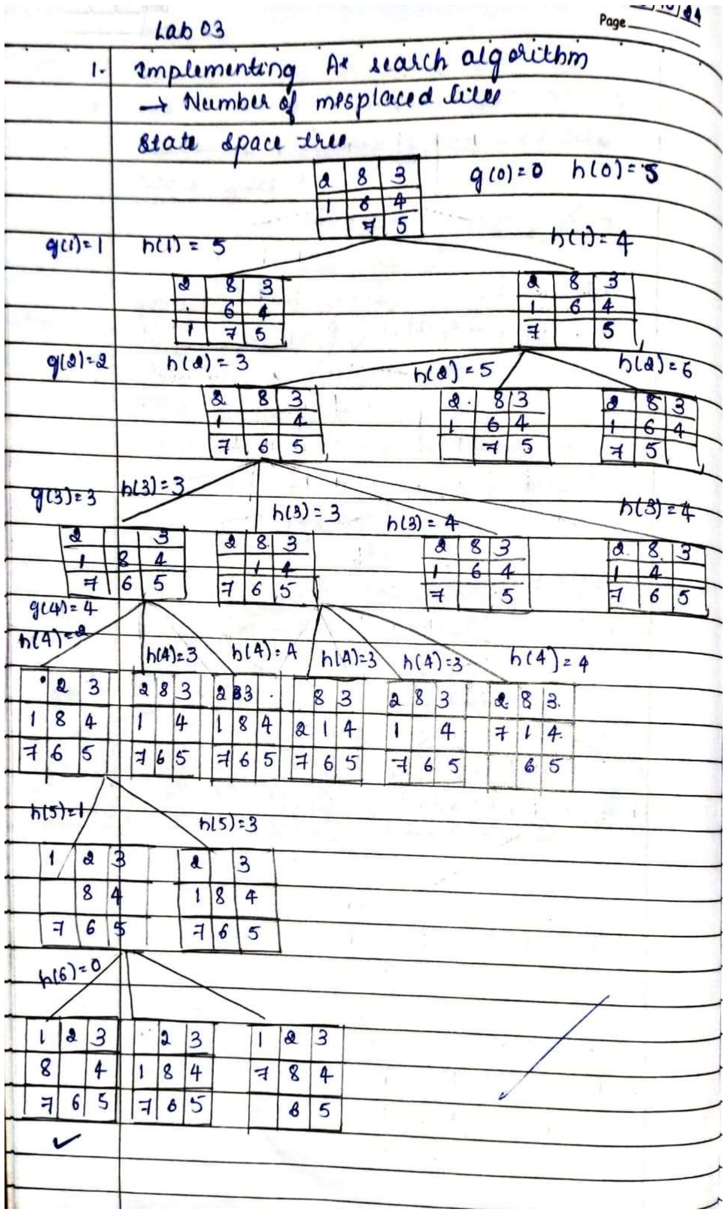
```
function solve_puzzle(initial_state, goal_state):
    priority_queue = [(heuristic(initial_state,
    goal_state), 0, initial_state, [])]
    visited = empty set
    while priority_queue is not empty:
        (f_cost, g_cost, current_state, current_path)
            = remove element with lowest f cost
              from priority_queue
        if current_state is equal to goal_state:
            return current_path + [current_state]
        if current_state is in visited:
            continue
        and current_state to visited.
        for next_state, action in get_possible_
        moves (current_state):
            new_g_cost = g_cost + 1
            new_f_cost = new_g_cost + heuristic
            (next_state, goal_state)
            add (new_f_cost, new_g_cost, next_state,
              current_path + [(current_state, action)]
              to priority_queue.
    return None.
function heuristic (state, goal_state):
    misplaced_tiles = 0
    for each tile in state:
        if tile is not in its correct position
        in goal_state and tile is not 0:
            misplaced_tiles = misplaced_tiles + 1
    return misplaced_tiles
function get_possible_moves (state):
    find position of the blank tile (0)
```

Page

```
    possible_moves = empty list
    if blank tile can move up:
        swap blank tile with tile above
        add (new_state, "Up") to possible_moves
    if blank tile can move down:
        swap blank tile with tile below
        add (new_state, "Down") to possible
        moves
    if blank tile can move left:
        swap blank tile with tile to the left
        add (new_state, "Left") to possible_moves
    if blank tile can move right:
        swap blank tile with tile to the right
        add (new_state, "Right") to possible_
        moves.
    return possible_moves.
```

15/10

State Space Diagram:



6. Implementing A* Algorithm(Manhattan Approach)

Code:

```
#Manhattan approach
import heapq
def solve_8puzzle(initial_state):
    goal_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
    priority_queue = [(heuristic(initial_state, goal_state), 0, initial_state, [])]
    visited = set()
    while priority_queue:
        f_cost, g_cost, current_state, current_path = heapq.heappop(priority_queue)
        if current_state == goal_state:
            return current_path + [current_state]
        if tuple(map(tuple, current_state)) in visited:
            continue
        visited.add(tuple(map(tuple, current_state)))
        for next_state, action in get_possible_moves(current_state):
            new_g_cost = g_cost + 1
            new_f_cost = new_g_cost + heuristic(next_state, goal_state)
            heapq.heappush(priority_queue, (new_f_cost, new_g_cost, next_state, current_path + [(current
_state, action)]))
    return None
def heuristic(state, goal_state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_row, goal_col = find_position(goal_state, state[i][j])
                distance += abs(i - goal_row) + abs(j - goal_col)
    return distance
def find_position(state, tile):
    for i in range(3):
        for j in range(3):
            if state[i][j] == tile:
                return i, j
def get_possible_moves(state):
    row, col = find_position(state, 0)
    possible_moves = []
    if row > 0:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row - 1][col] = new_state[row - 1][col], new_state[row][col]
        possible_moves.append((new_state, 'Up'))
    if row < 2:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row + 1][col] = new_state[row + 1][col], new_state[row][col]
        possible_moves.append((new_state, 'Down'))
    if col > 0:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row][col - 1] = new_state[row][col - 1], new_state[row][col]
        possible_moves.append((new_state, 'Left'))
    if col < 2:
        new_state = [list(row) for row in state]
        new_state[row][col], new_state[row][col + 1] = new_state[row][col + 1], new_state[row][col]
```

```
        possible_moves.append((new_state, 'Right'))
    return possible_moves
initial_state = [[2, 8, 3], [1, 6, 4], [0, 7, 5]]
solution = solve_8puzzle(initial_state)
if solution:
    print("Solution found:")
    for state, action in solution[:-1]:
        print("--------------------")
        for row in state:
            print(row)
        print("Move:", action)
    print("--------------------")
    for row in solution[-1]:
        print(row)
else:
    print("No solution found.")
```

Output:

```
Solution found:
--------------------
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]
Move: Right
--------------------
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Move: Up
--------------------
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Move: Up
--------------------
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Move: Left
--------------------
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
Move: Down
--------------------
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
Move: Right
--------------------
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
```

Pseudo Code:

```
Pseudocode for Manhatten Distance
function solve_8puzzle(initialstate, goal
state):
    priority_queue = [(manhattan_distance(init
    -ial_state, goal_state), 0, initial_state, [])]
    visited = empty set
    while priority_queue is not empty:
        (f-cost, g-cost, currentstate, current_path)
        = pop smallest f-cost from queue
        if current_state == goalstate: return current
        path + [current_state]
        if current_state in visited: continue
        visited.add(current_state)
        for next_state, action in get_moves(current_sta
        -te):
            new g-cost = g-cost + 1
            new_f-cost = new g-cost + mahattan_distance
            (next state, goal_state)
            add (new_f cost, new g-cost, next_state,
            current_path + [(current_state, action)])
            to queue
    return None # No solution.
function manhattan_distance(state, goal_state):
    distance = 0
    for each tile in state:
        if tile != 0:
            distance += |goal row - current_row| +
            |goalcol - current col|
    return distance
function get_moves(state):
    find blank tile position.
    generate possible moves (Up, Down, Left, Right)
    by swapping blank tile with adjacent tiles.
    return list of (new_state, action) pairs.
```

State Space Diagram :