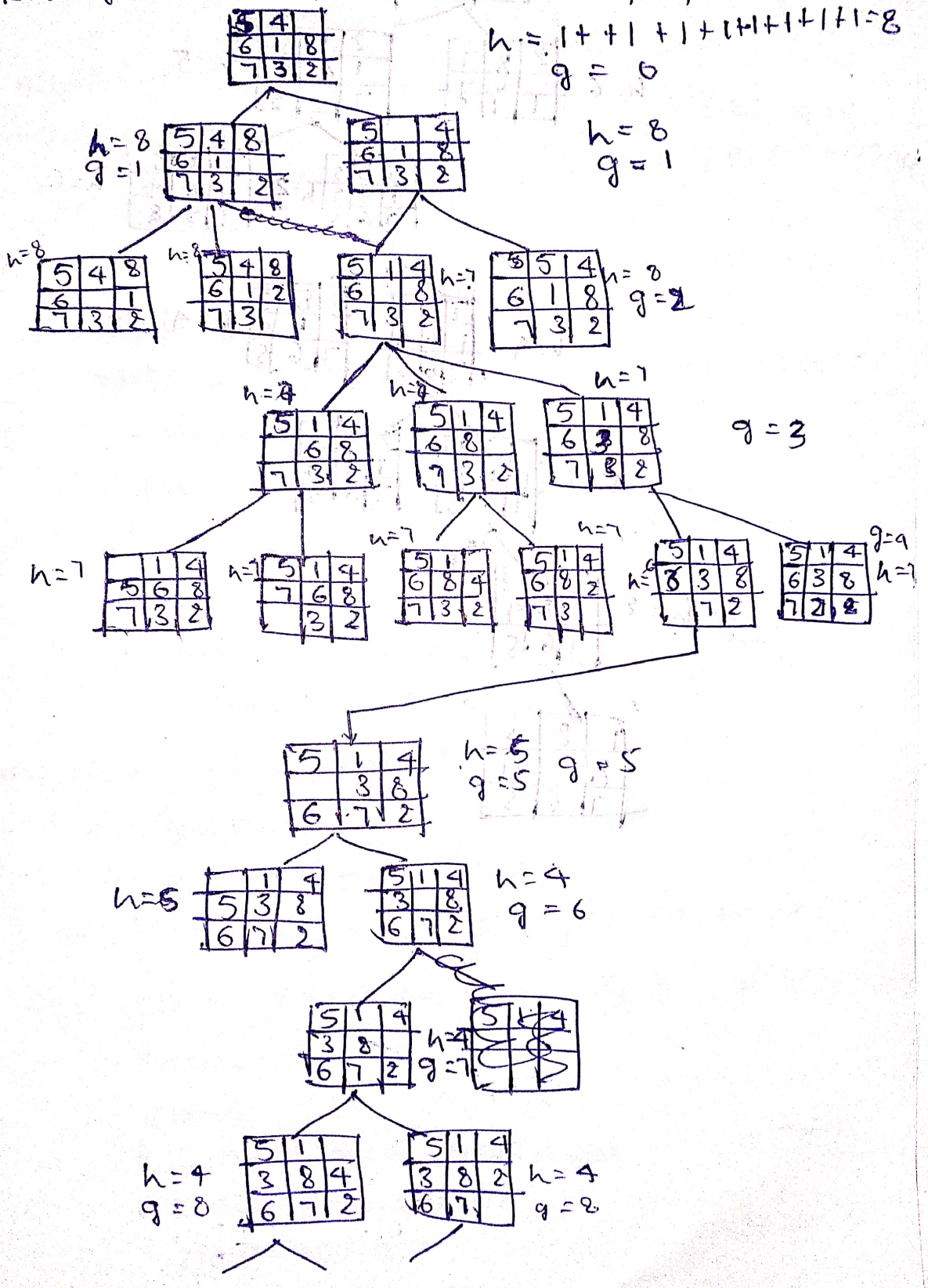
Implement A* search algorithm 8 puzzle problem

5	4	
6	1	8
7	3	2

	1	2
3	4	5
6	7	8

1) Heuristic : Number of tiles out of place

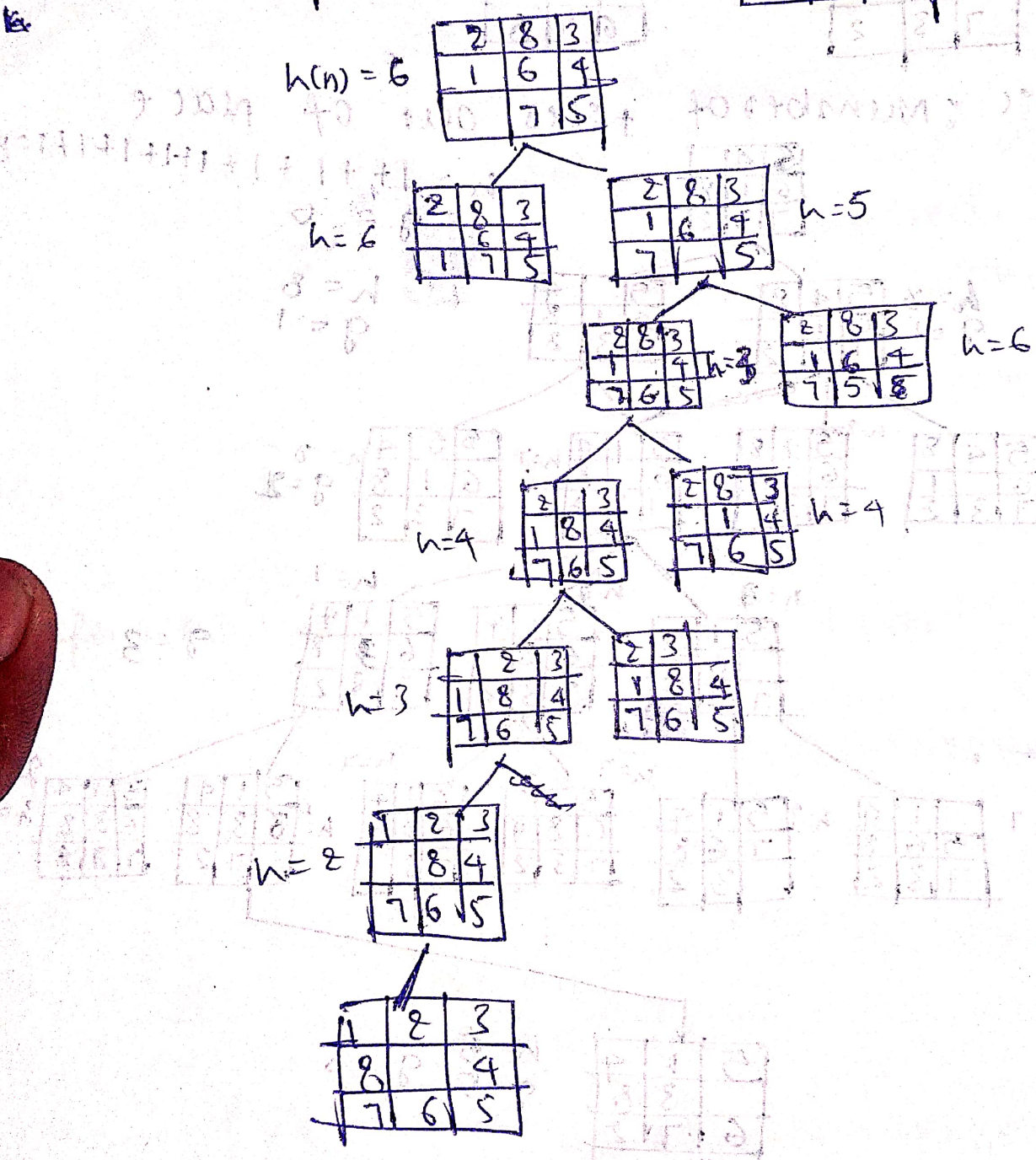


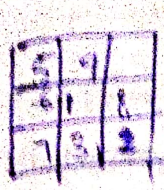
Initial

2	8	3
1	6	4
	7	5

goal state

1	2	3
8		4
7	6	5





Algorithm:

```
class puzzlestate:
    function __init__(board, zero_pos, g=0,
                      parent=None):
        this.board = board
        this.zero_pos = zero_pos
        this.g = g
        this.h = calculate_misplaced_tiles()
        this.f = g + h
        this.parent = parent
    function calculate_misplaced_tiles():
        goal = [0, 1, 2, 3, 4, 5, 6, 7, 8]
        return count of tiles not in
               goal position
    function get_neighbors():
        neighbors = []
        row, col = this.zero_pos
        directions = [(-1,0), (1,0), (0,-1), (0,1)]
        for each direction (dr, dc) in directions:
            new_row = row + dr
            new_col = col + dc
            if (new_row, new_col) is within
            board boundaries:
                new_board = copy of this.board
                Swap zero tile with neighbor tile
                neighbors.append(puzzlestate(new_board,
                    (new_row, new_col), this.g+1,
                    this))
        return neighbors
```

```
function __lt__(other):
    return this.f < other.f
function a-star(initial_board)
    zero_pos = index of 0 in initial_board
    initial_state = PuzzleState(initial_board, zero_pos)
    open_set = priority queue initialized with
               initial_state
    closed_set = empty set

    while open_set is not empty:
        current_state = pop the state with
                        lowest f from open_set
        if current_state.h == 0:
            print("Solution found:")
            print_path(current_state)
            return current_state.g
        add current_state.board to closed_set

        for each neighbor in current_state.get_
                                    neighbors():
            if neighbor.board is in closed_set:
                continue
            if neighbor.board is not in open_set:
                add neighbor to open_set

    return None.
function get_board_input(prompt):
    while True:
        try:
            board = read input and convert to
                    list of integers
```

```
if length of board is not 9 or board does
not contain unique integers from 0 to 8:
    raise valueError
return board
except valueError
    print ("Invalid Input")

main:
    initial_board = get_board_input("Enter Initial
    goal_board = get_board_input("Enter goal")

    set pazzlestate.goal_state = goal_board

    steps = a_star(initial_board)
    if steps is not NO:
        print("solution", steps, "moves")
    else:
        print("NO solution")
```

Sun
15/10/24.

Manhatten Distance

Algorithm:

```
FUNCTION A_STAR(initial_state):
    priority_queue = empty min-heap
    visited = empty set

    h = MANHATTAN_DISTANCE(initial_state)
    g = 0
    f = g + h
HEAP_PUSH(priority_queue, (f, initial_state, [], 0))
WHILE priority queue is not empty:
    (f, current_state, path, g) = HEAP_POP(priority_queue)

    PRINT "Exploring state in A*:"
    PRINT_STATE(current_state)

IF IS_GOAL(current_state)
    RETURN path

ADD tuple(current_state) to visited

FOR direction IN ["up", "down", "left", "right"]:
    new_state = MOVE(current_state, direction)

    IF new_state is not None AND tuple(new_state)
        not in visited

        new_g = g + 1
        h = MANHATTAN_DISTANCE(new_state)
        new_f = new_g + h

        HEAP_PUSH(priority_queue, (new_f, new_state,
                                   path + [direction], new_g))

RETURN None
```

```
FUNCTION MANHATTAN-DISTANCE (state):
    distance=0
    FOR i FROM 0 TO 2:
        FOR j FROM 0 TO 2:
            IF state[i][j] != 0
            goal_i, goal_j = DIVMOD(state[i][j]-1,3)
            IF state[i][j] == 8
            goal_i, goal_j = 1, 1
            distance += ABS(goal_i - i) + ABS(goal_j - j)
    RETURN distance

FUNCTION IS_GOAL(state):
    RETURN state == goal_state

FUNCTION PRINT_STATE(state)
    FOR row IN state:
        PRINT row
    print "\n"

FUNCTION MOVE(state, direction):
    new_state = COPY(state)
    zero_pos = FIND_ZERO(state)
    i, j = zero_pos
    IF direction == "up" AND i>0:
        SWAP(new_state[i][j], new_state[i-1][j])
    ELSE IF direction == "down" AND i<2:
        SWAP(new_state[i][j], new_state[i+1][j])
    ELSE IF direction == "left" AND j>0:
        SWAP(new_state[i][j], new_state[i][j-1])
    ELSE IF direction == "right" AND j<2
        SWAP(new_state[i][j], new_state[i][j+1])
    ELSE:
        RETURN None
```

```
RETURN new-state

FUNCTION FIND_ZERO(state):
    FOR i FROM 0 TO 2:
        FOR j FROM 0 to 2:
            IF state[i][j] == 0:
                RETURN (i, j)
```