**LAB 4:**

**Program:** A\* Algorithm with Misplaced Tiles Heuristic

**Code:**

import heapq

class PuzzleState:

    def \_\_init\_\_(self, board, zero\_pos, moves=0):

        self.board = board

        self.zero\_pos = zero\_pos

        self.moves = moves

        self.heuristic = self.calculate\_misplaced\_tiles()

    def calculate\_misplaced\_tiles(self):

        goal\_state = [[1, 2,3], [8, 0, 4], [7, 6, 5]]

        return sum(1 for i in range(3) for j in range(3) if self.board[i][j] != goal\_state[i][j])

    def get\_neighbors(self):

        neighbors = []

        x, y = self.zero\_pos

        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        for dx, dy in directions:

            new\_x, new\_y = x + dx, y + dy

            if 0 <= new\_x < 3 and 0 <= new\_y < 3:

                new\_board = [row[:] for row in self.board]

                new\_board[x][y], new\_board[new\_x][new\_y] = new\_board[new\_x][new\_y], new\_board[x][y]

                neighbors.append((new\_board, (new\_x, new\_y)))

        return neighbors

    def \_\_lt\_\_(self, other):

        return (self.moves + self.heuristic) < (other.moves + other.heuristic)

    def \_\_repr\_\_(self):

        return f"{self.board} (Moves: {self.moves}, Heuristic: {self.heuristic})"

def a\_star(initial\_board):

    initial\_zero\_pos = next((i, j) for i in range(3) for j in range(3) if initial\_board[i][j] == 0)

    initial\_state = PuzzleState(initial\_board, initial\_zero\_pos)

    priority\_queue = []

    heapq.heappush(priority\_queue, initial\_state)

    visited = set()

    while priority\_queue:

        current\_state = heapq.heappop(priority\_queue)

        print(f"Exploring State: {current\_state}")

        if current\_state.heuristic == 0:

            print(f"Reached goal in {current\_state.moves} moves.")

            return

        visited.add(tuple(map(tuple, current\_state.board)))

        for neighbor\_board, neighbor\_zero\_pos in current\_state.get\_neighbors():

            neighbor\_state = PuzzleState(neighbor\_board, neighbor\_zero\_pos, current\_state.moves + 1)

            if tuple(map(tuple, neighbor\_board)) not in visited:

                heapq.heappush(priority\_queue, neighbor\_state)

    print("No solution found.")

if \_\_name\_\_ == "\_\_main\_\_":

    initial\_board = [

        [2, 8, 3],

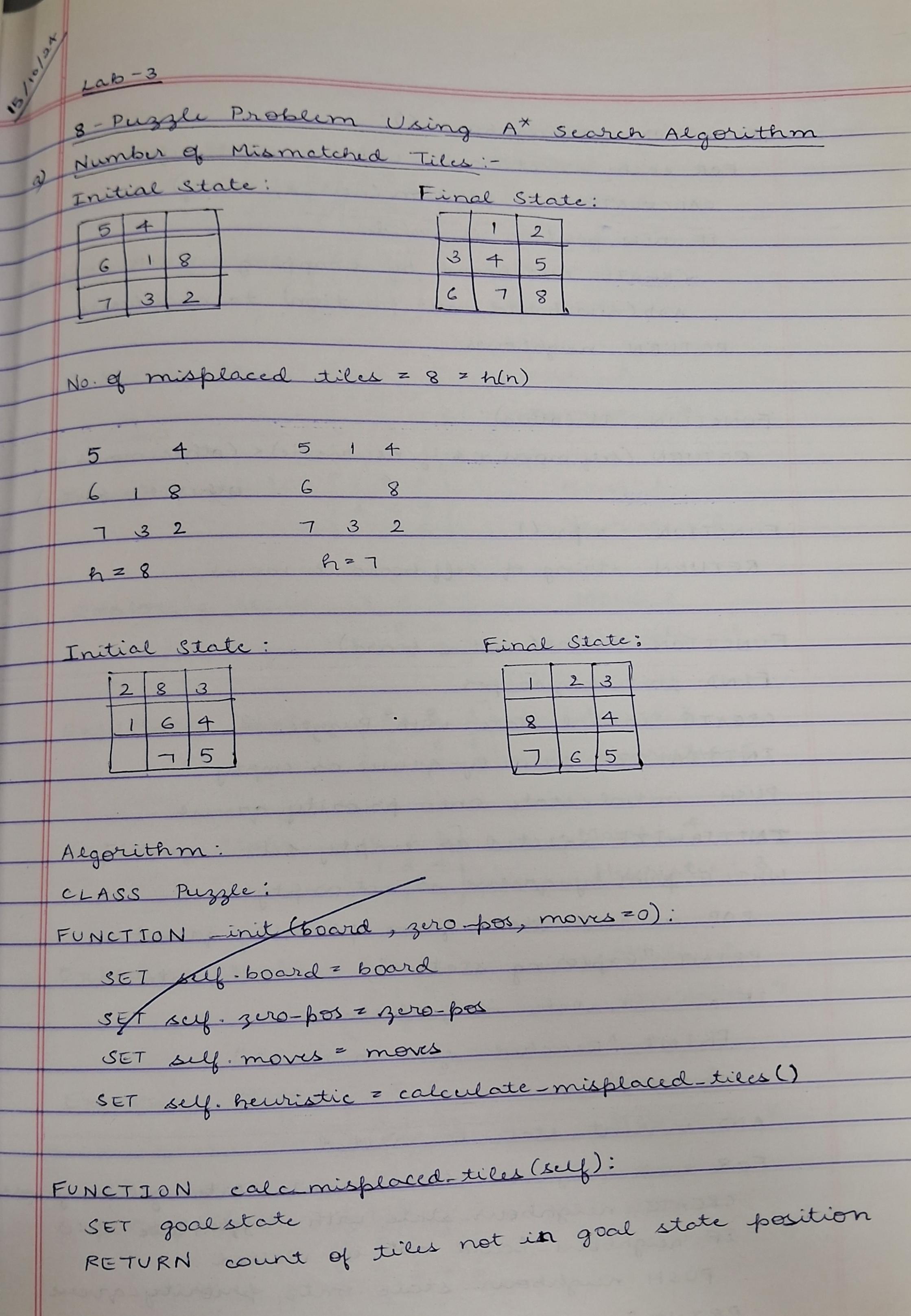
        [1, 6, 4],

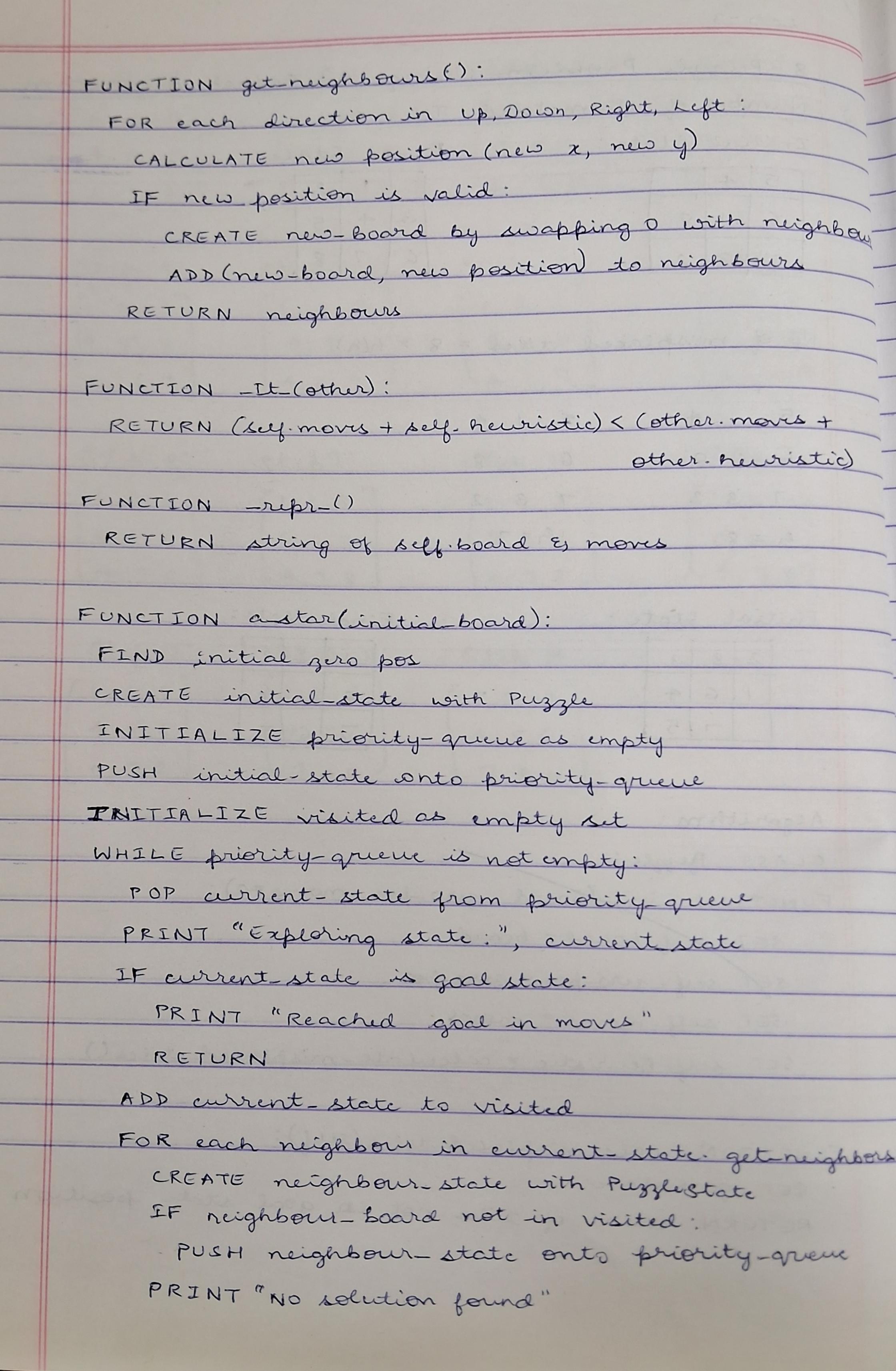
        [0, 7, 5]

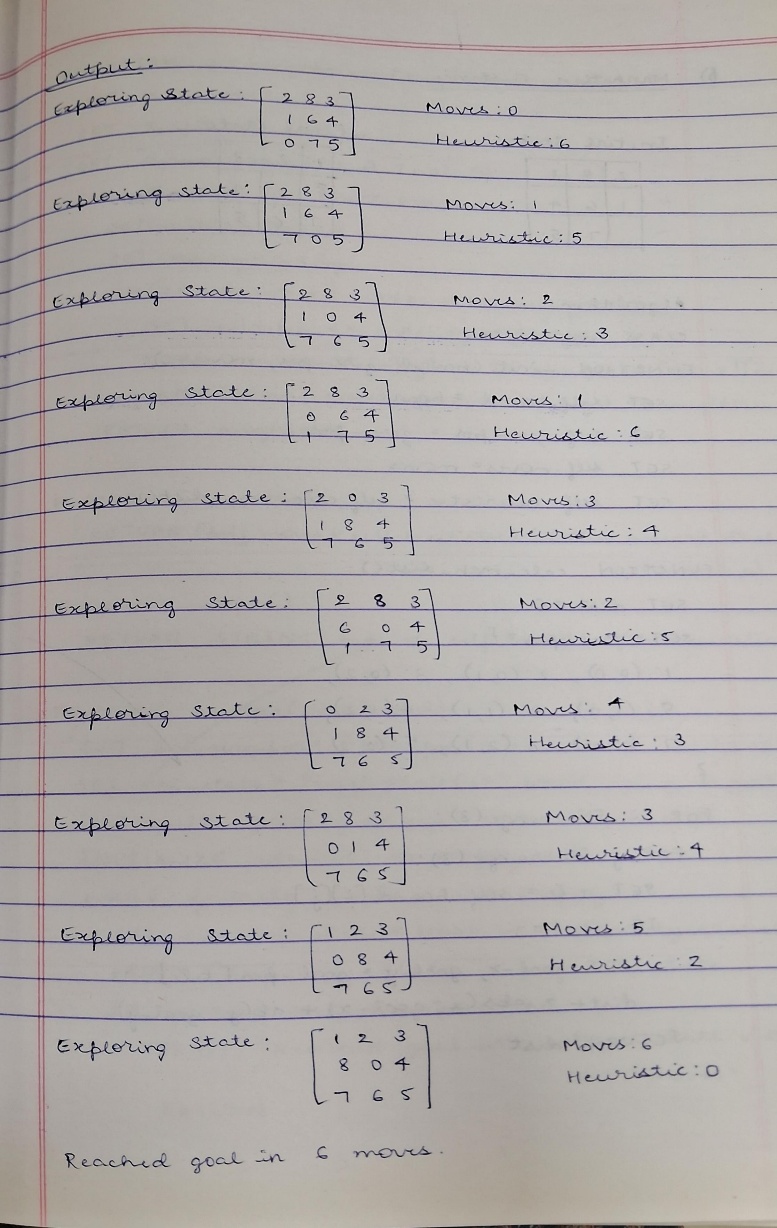
    ]

a\_star(initial\_board)

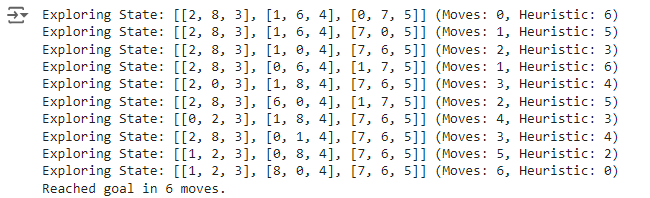
**Algorithm:**







**Output:**

****

### Program: A\* Search in 8-Puzzle using Manhattan

**Code:**

import heapq

class PuzzleState:

    def \_\_init\_\_(self, board, zero\_pos, moves=0):

        self.board = board

        self.zero\_pos = zero\_pos

        self.moves = moves

        self.heuristic = self.calculate\_manhattan\_distance()

    def calculate\_manhattan\_distance(self):

        distance = 0

        goal\_positions = {

            1: (0, 0), 2: (0, 1), 3: (0, 2),

            8: (1, 0), 0: (1, 1), 4: (1, 2),

            7: (2, 0), 6: (2, 1), 5: (2, 2)

        }

        for i in range(3):

            for j in range(3):

                tile = self.board[i][j]

                if tile != 0:

                    goal\_x, goal\_y = goal\_positions[tile]

                    distance += abs(i - goal\_x) + abs(j - goal\_y)

        return distance

    def get\_neighbors(self):

        neighbors = []

        x, y = self.zero\_pos

        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        for dx, dy in directions:

            new\_x, new\_y = x + dx, y + dy

            if 0 <= new\_x < 3 and 0 <= new\_y < 3:

                new\_board = [row[:] for row in self.board]

                new\_board[x][y], new\_board[new\_x][new\_y] = new\_board[new\_x][new\_y], new\_board[x][y]

                neighbors.append((new\_board, (new\_x, new\_y)))

        return neighbors

    def \_\_lt\_\_(self, other):

        return (self.moves + self.heuristic) < (other.moves + other.heuristic)

    def \_\_repr\_\_(self):

        return f"{self.board} (Moves: {self.moves}, Heuristic: {self.heuristic})"

def a\_star(initial\_board):

    initial\_zero\_pos = next((i, j) for i in range(3) for j in range(3) if initial\_board[i][j] == 0)

    initial\_state = PuzzleState(initial\_board, initial\_zero\_pos)

    priority\_queue = []

    heapq.heappush(priority\_queue, initial\_state)

    visited = set()

    while priority\_queue:

        current\_state = heapq.heappop(priority\_queue)

        print(f"Exploring State: {current\_state}")

        if current\_state.heuristic == 0:

            print(f"Reached goal in {current\_state.moves} moves.")

            return

        visited.add(tuple(map(tuple, current\_state.board)))

        for neighbor\_board, neighbor\_zero\_pos in current\_state.get\_neighbors():

            neighbor\_state = PuzzleState(neighbor\_board, neighbor\_zero\_pos, current\_state.moves + 1)

            if tuple(map(tuple, neighbor\_board)) not in visited:

                heapq.heappush(priority\_queue, neighbor\_state)

    print("No solution found.")

if \_\_name\_\_ == "\_\_main\_\_":

    initial\_board = [

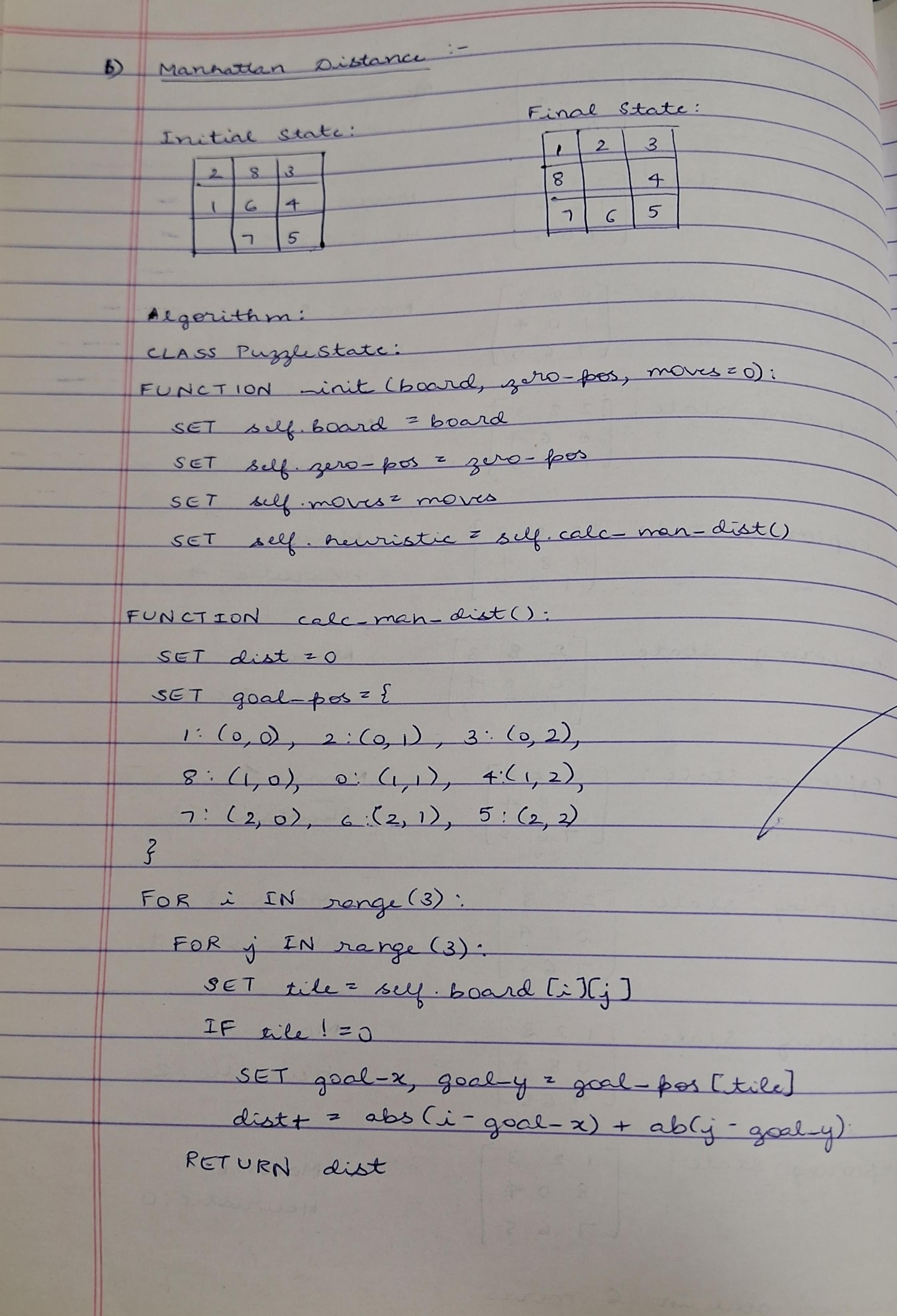
        [2, 8, 3],

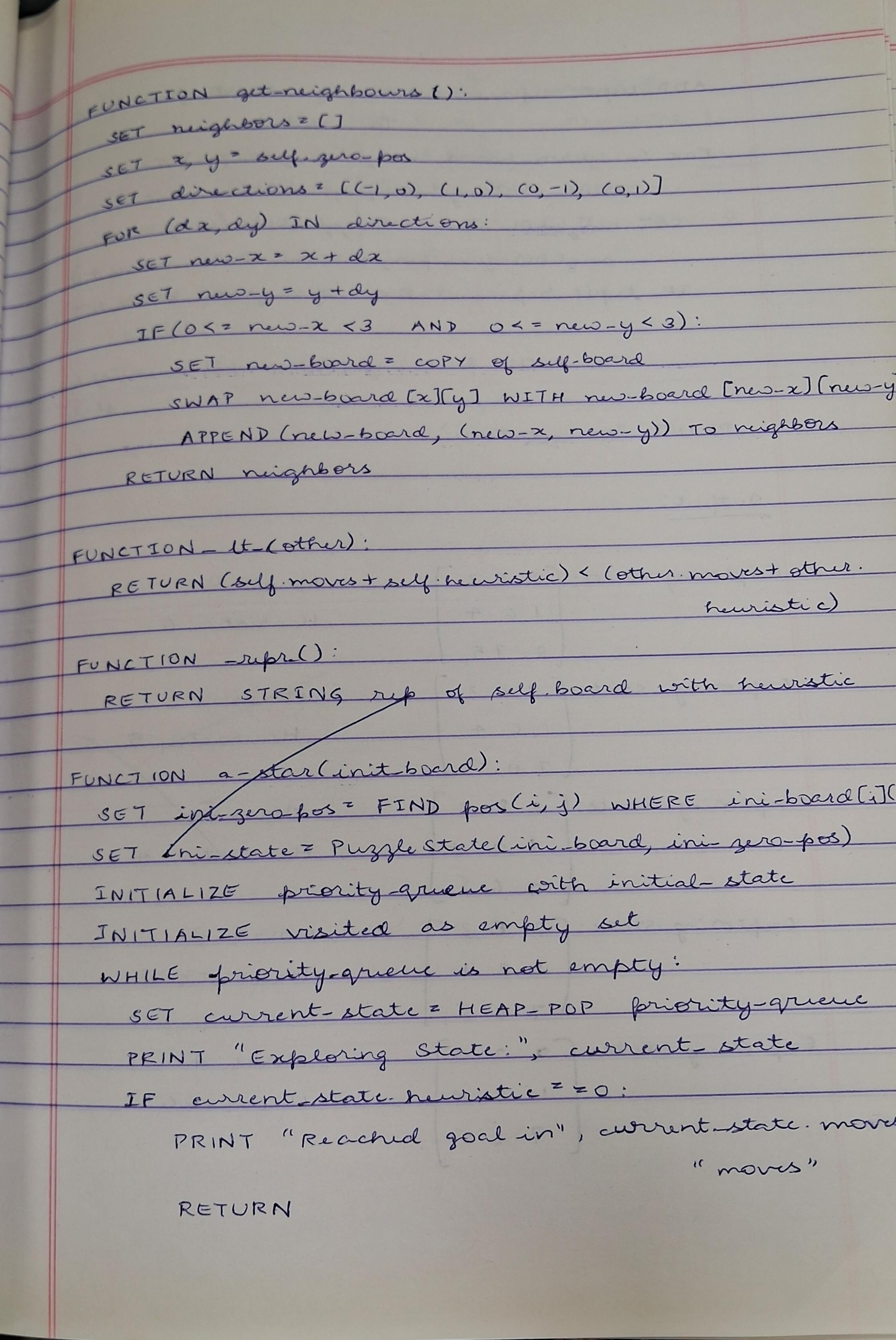
        [1, 6, 4],

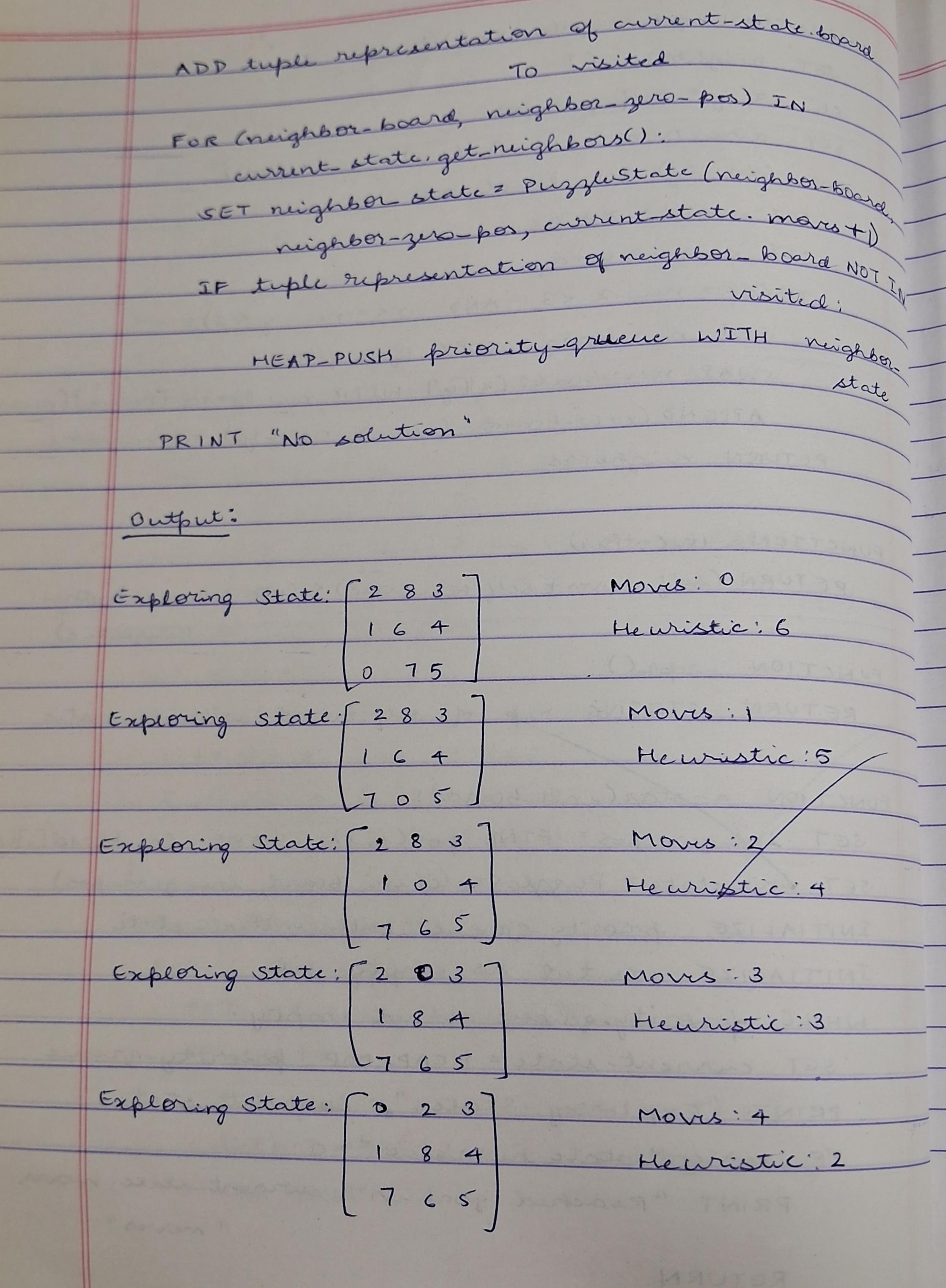
        [0, 7, 5]

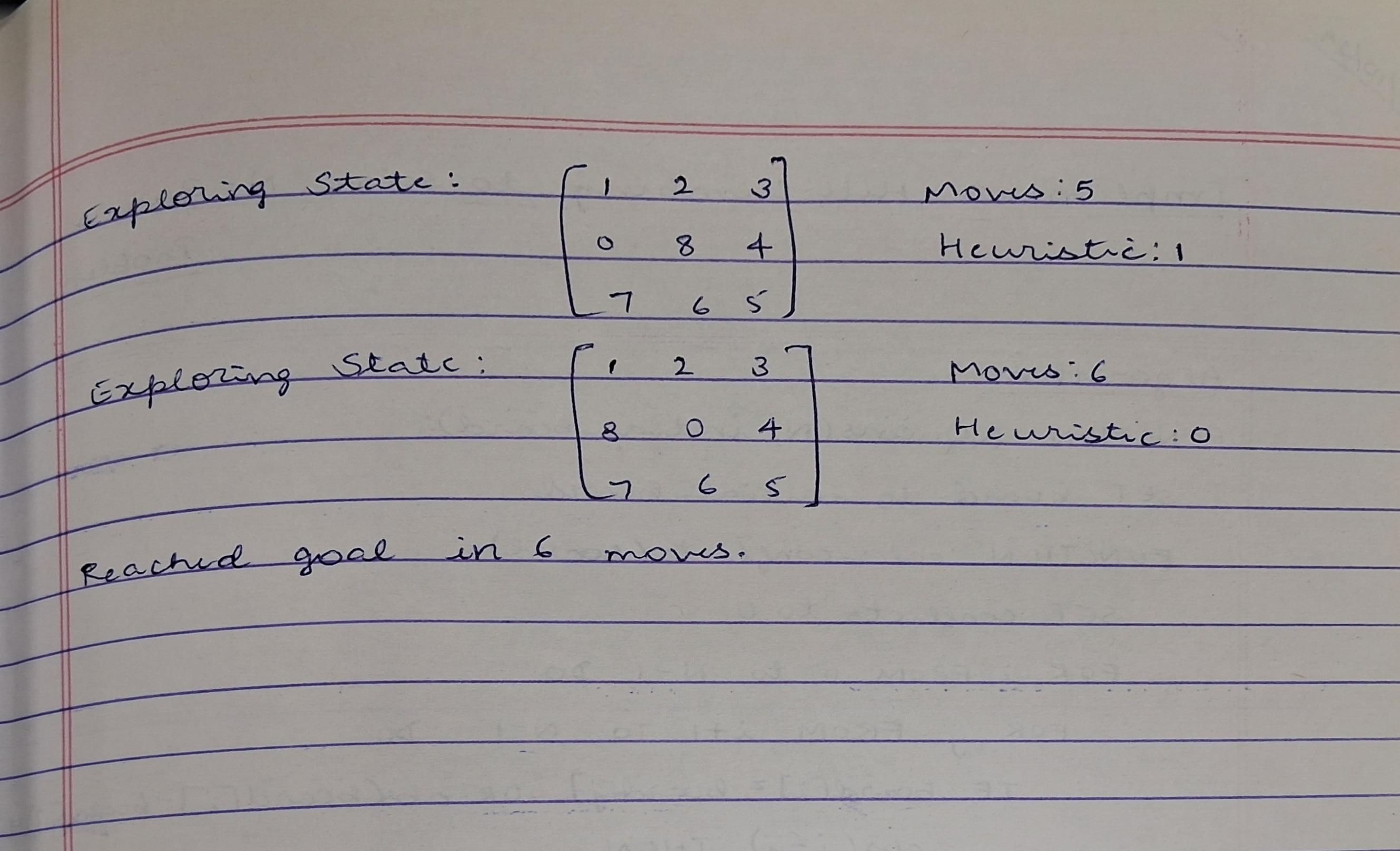
    ]

    a\_star(initial\_board)

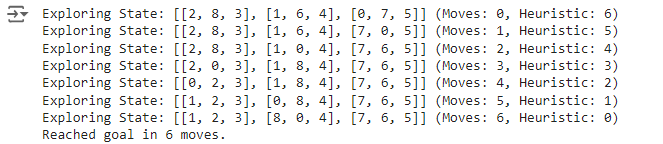
**Algorithm:**

****

****

****

**Output:**

****