Sumedh ahire
FYMCA-B 03
BATCH 1
ASSIGNEMT 2

CODE:

```
from collections import deque

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


class State:
  def __init__(self, board, parent=None):
    self.board = board
    self.parent = parent

  def __hash__(self):
    return hash(tuple(self.board))

  def __eq__(self, other):
    return self.board == other.board

  def __str__(self):
    board_str = ""
    for i in range(3):
      for j in range(3):
        board_str += str(self.board[i * 3 + j]) + " "
      board_str += "\n"
    return board_str

  def get_empty_tile_position(self):
    for i in range(9):
      if self.board[i] == 0:
        return i // 3, i % 3

  def get_valid_moves(self):
    empty_row, empty_col = self.get_empty_tile_position()
    moves = []
    if empty_row > 0:  # Up move possible
      moves.append((empty_row - 1, empty_col))
    if empty_row < 2:  # Down move possible
      moves.append((empty_row + 1, empty_col))
    if empty_col > 0:  # Left move possible
      moves.append((empty_row, empty_col - 1))
    if empty_col < 2:  # Right move possible
      moves.append((empty_row, empty_col + 1))
    return moves

  def generate_successors(self):
    successors = []
    empty_row, empty_col = self.get_empty_tile_position()
    for new_row, new_col in self.get_valid_moves():
      new_board = self.board.copy()
      new_board[empty_row * 3 + empty_col] = new_board[new_row * 3 + new_col]
```

```
      new_board[new_row * 3 + new_col] = 0
      successors.append(State(new_board, self))
    return successors


def heuristic(state):
  # Manhattan distance heuristic: sum of distances of each tile from its goal position
  distance = 0
  for i, tile in enumerate(state.board):
    goal_row, goal_col = divmod(GOAL_STATE.index(tile), 3)
    current_row, current_col = divmod(i, 3)
    distance += abs(current_row - goal_row) + abs(current_col - goal_col)
  return distance


def a_star_search(initial_state):
  open_set = deque([initial_state])
  closed_set = set()
  g_score = {initial_state: 0}
  f_score = {initial_state: heuristic(initial_state)}

  while open_set:
    current = min(open_set, key=lambda state: f_score[state])
    open_set.remove(current)
    closed_set.add(current)

    if current.board == GOAL_STATE:
      # Goal state reached
      path = []
      while current:
        path.append(current)
        current = current.parent
      return path[::-1]  # Reverse path to get order of moves

    for successor in current.generate_successors():
      if successor in closed_set:
        continue

      tentative_g_score = g_score[current] + 1

      if successor not in open_set or tentative_g_score < g_score.get(successor, float('inf')):
        g_score[successor] = tentative_g_score
        f_score[successor] = tentative_g_score + heuristic(successor)
        open_set.append(successor)

  return None  # No solution found

# Initial state (replace with your desired starting configuration)
initial_state = State([1, 8, 2, 0, 4, 3, 7, 6, 5])

solution = a_star_search(initial_state)
OUTPUT:
```

```
1  8  2
0  4  3
7  6  5

# Move 1 (swap empty tile with tile above)
1  8  2
4  0  3
7  6  5

# Move 2 (swap empty tile with tile to the left)
8  1  2
4  0  3
7  6  5

# ... (continues showing subsequent moves)

# Goal state reached
1  2  3
8  0  4
7  6  5
```