Aim:

To write and execute the python program for the sliding puzzle problem.

Procedure:

1. **Initialize Puzzle:**
   * The program defines a **SlidingPuzzle** class to represent the sliding puzzle game.
   * The puzzle starts with tiles **[1, 2, 3, 4, 5, 6, 7, 8, None]**.
   * The tiles are shuffled randomly to create an initial game state.
2. **Display and Move Tiles:**
   * The puzzle can be displayed in a 3x3 grid using the **display()** method.
   * The user is prompted to input a number (1-8) corresponding to the tile they want to move.
   * The program checks if the selected tile can be moved based on its position relative to the empty space (None).
3. **Move Validation:**
   * If the selected tile can be moved (adjacent to the empty space), the puzzle state is updated.
   * If the move is not valid, an error message is displayed.
4. **Repeat Until Solved:**
   * The user continues making moves until the puzzle is solved.
   * The puzzle is considered solved when the tiles are in ascending order followed by the empty space (None).
5. **Congratulations Message:**
   * Once the puzzle is solved, the program prints a congratulatory message.
6. **Example Interaction:**
   * User sees the initial puzzle state and is prompted to enter a number to make a move.
   * The program displays the updated state after each move and informs the user if a move is invalid.
   * The process continues until the user successfully arranges the tiles in the correct order.

Code:

import heapq

def solve\_puzzle(initial):

goal = (1, 2, 3, 4, 5, 6, 7, 8, 0)

moves = [1, -1, 3, -3] # Simplified moves considering the flat list

open\_list, closed\_set = [(0, initial, [])], set()

while open\_list:

\_, current, path = heapq.heappop(open\_list)

if current == goal:

return path + [current]

closed\_set.add(current)

empty = current.index(0)

for m in moves:

if 0 <= empty // 3 + m // 3 < 3 and m + empty in range(9): # Check valid moves considering edges

neighbor = list(current)

neighbor[empty], neighbor[empty + m] = neighbor[empty + m], neighbor[empty]

neighbor\_tuple = tuple(neighbor)

if neighbor\_tuple not in closed\_set:

heapq.heappush(open\_list, (len(path) + 1, neighbor\_tuple, path + [current]))

return []

def print\_state(state):

for i in range(0, 9, 3):

print(state[i:i+3])

print('---')

initial\_state = (1, 0, 3, 4, 2, 5, 7, 8, 6)

solution\_path = solve\_puzzle(initial\_state)

if solution\_path:

print("Solution path:")

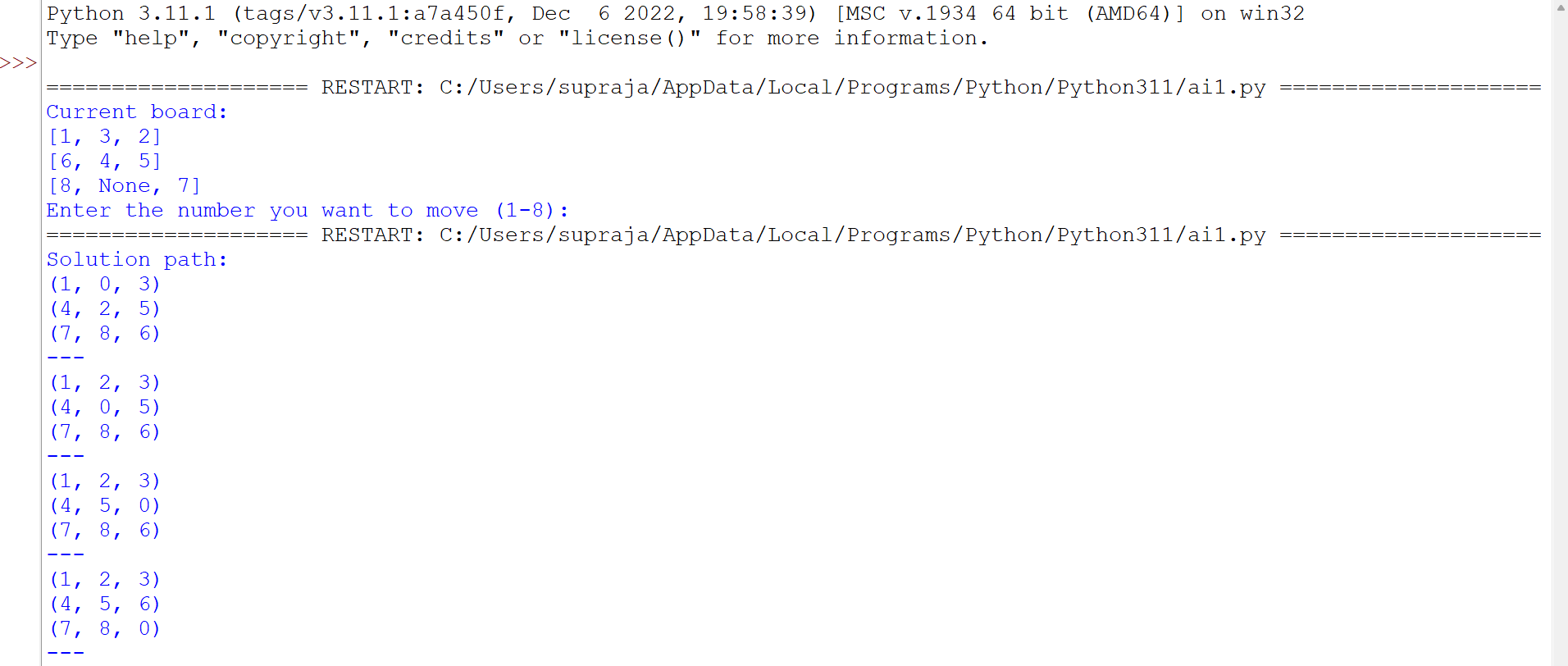
for step in solution\_path:

print\_state(step)

else:

print("No solution found.")

output:



Result:

Hence the program has been successfully executed and verified.