Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

Subject: Artificial Intelligence (DJ19DSC502)

AY: 2024-25

Experiment 1
(Problem Solving)

NAME: DHRUV SHAH SAPID: 60009220132

Aim: Implement domain specific functions for given problems required for problem solving.

Theory:

There are two domain specific functions required in all problem solving methods.
1. GoalTest Function:

goalTest(State) Returns *true* if the input state is the goal state and *false* otherwise.

goalTest(State, Goal) Returns *true* if *State* matches *Goal*, and *false* otherwise.

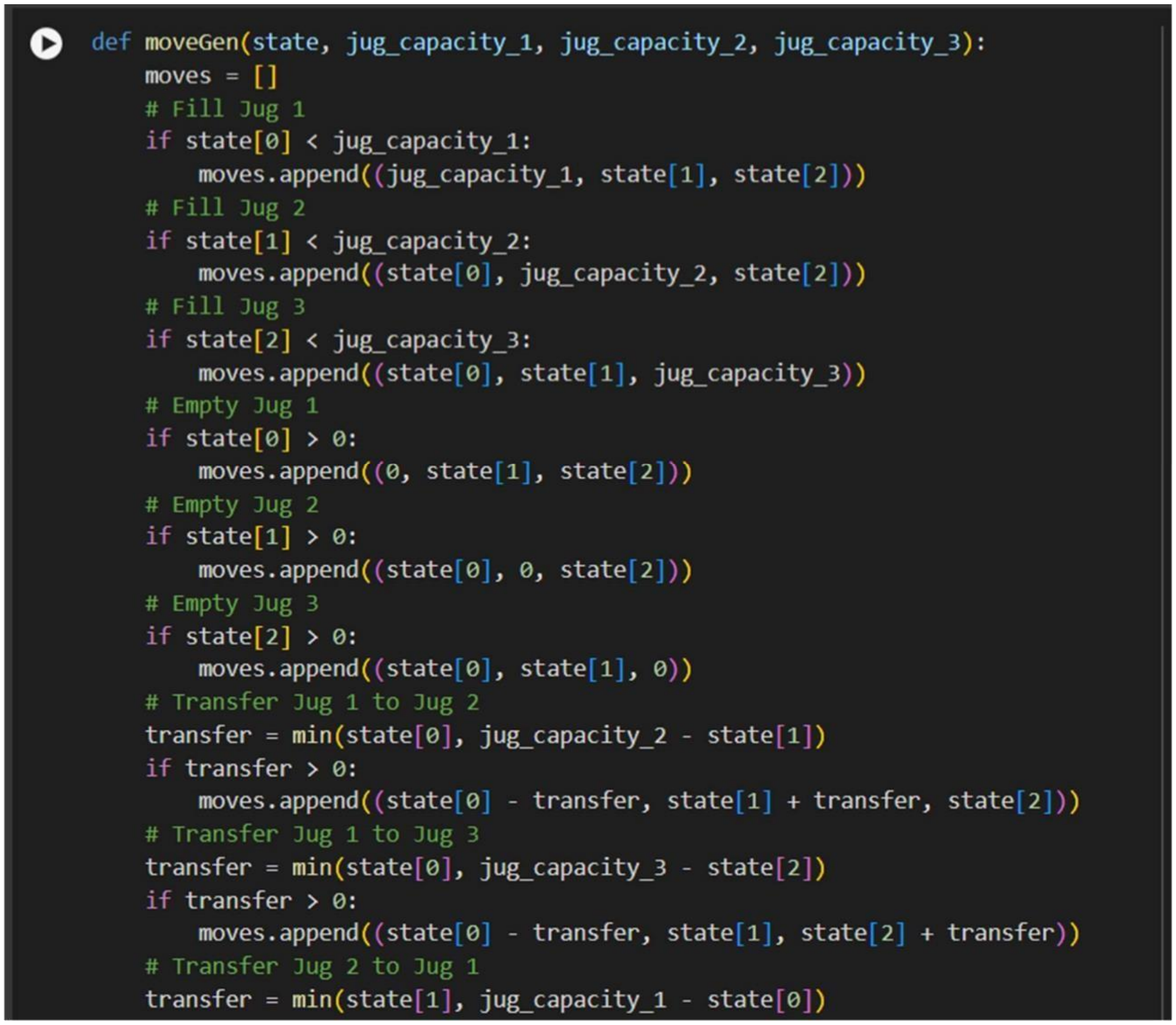
2. MoveGen function:

```
Initialize set of successors C to empty set.
Add M to the complement of given state N to get new state S.
If given state has Left, then add Right to S, else add Left.
If legal(S) then add S to set of successors C.
For each other-entity E in N
    make a copy S' of S,
    add E to S',
    If legal (S'), then add S' to C.
Return (C).
```

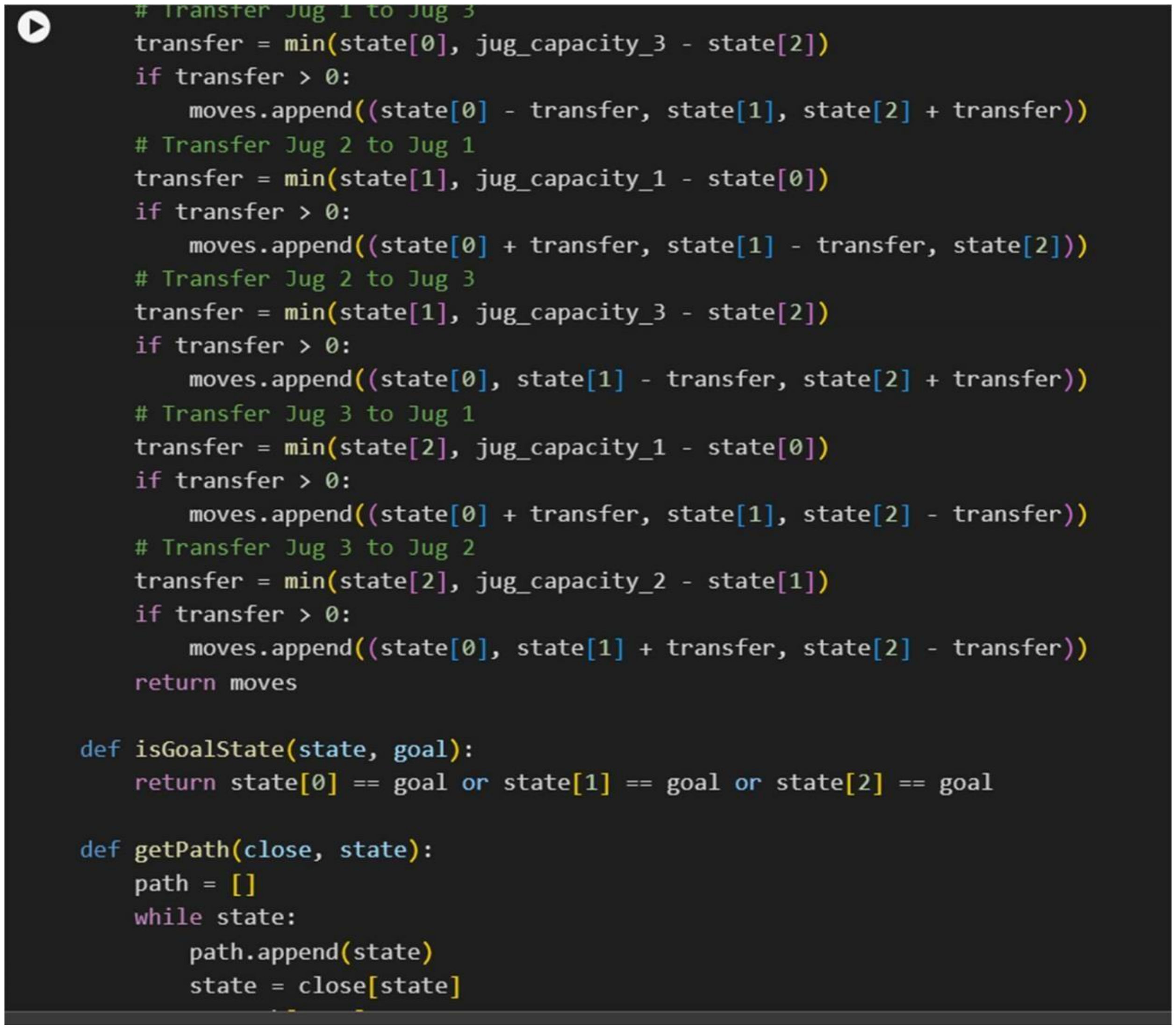
Lab Assignment to do:

Create MoveGen and GoalTest Functions for the given problems 1.
Water Jug Problem

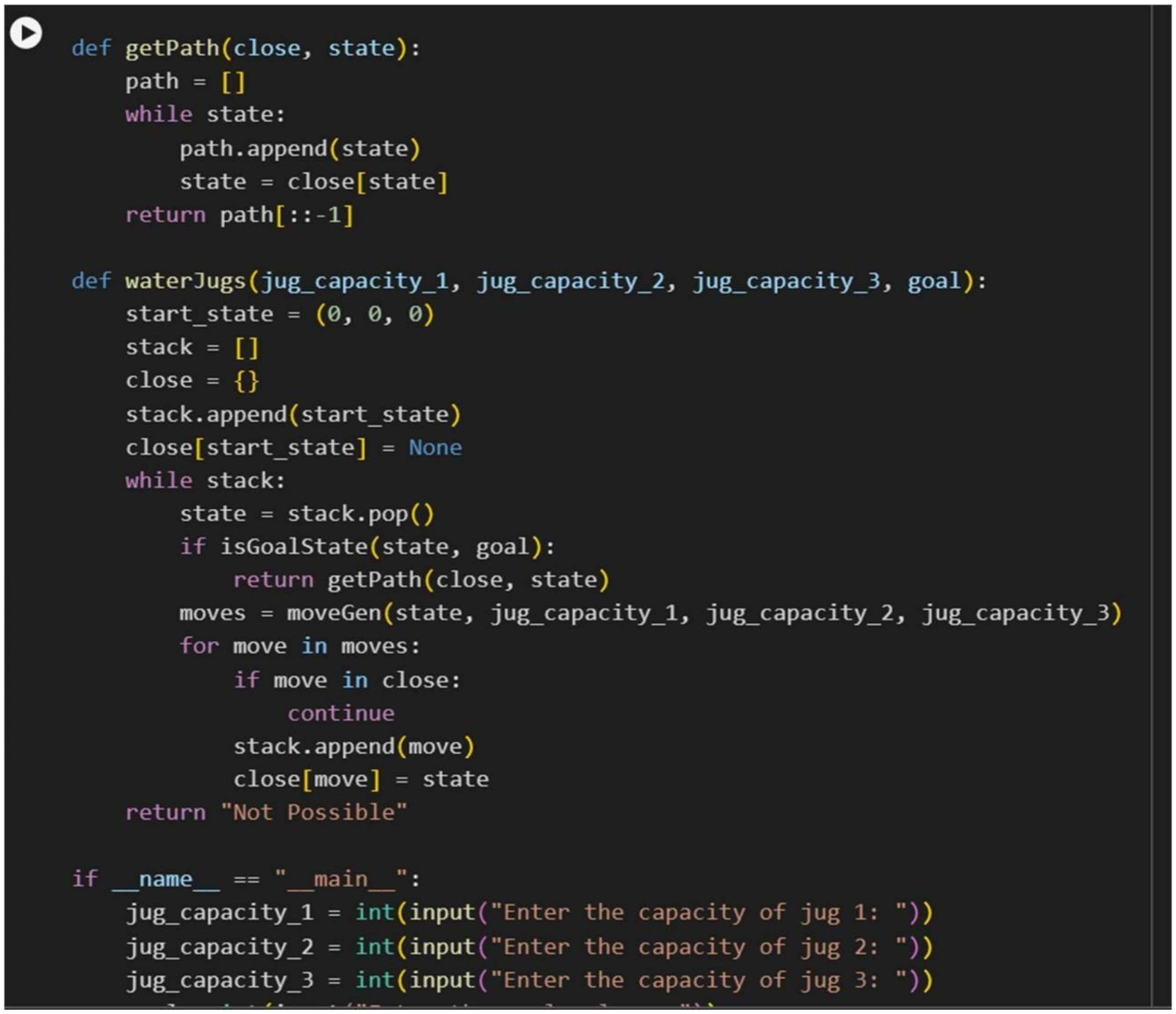
There are two jugs available of different volumes such as a 3 litres and a 7 litres and you have to measure a different volume such as 6 litre.



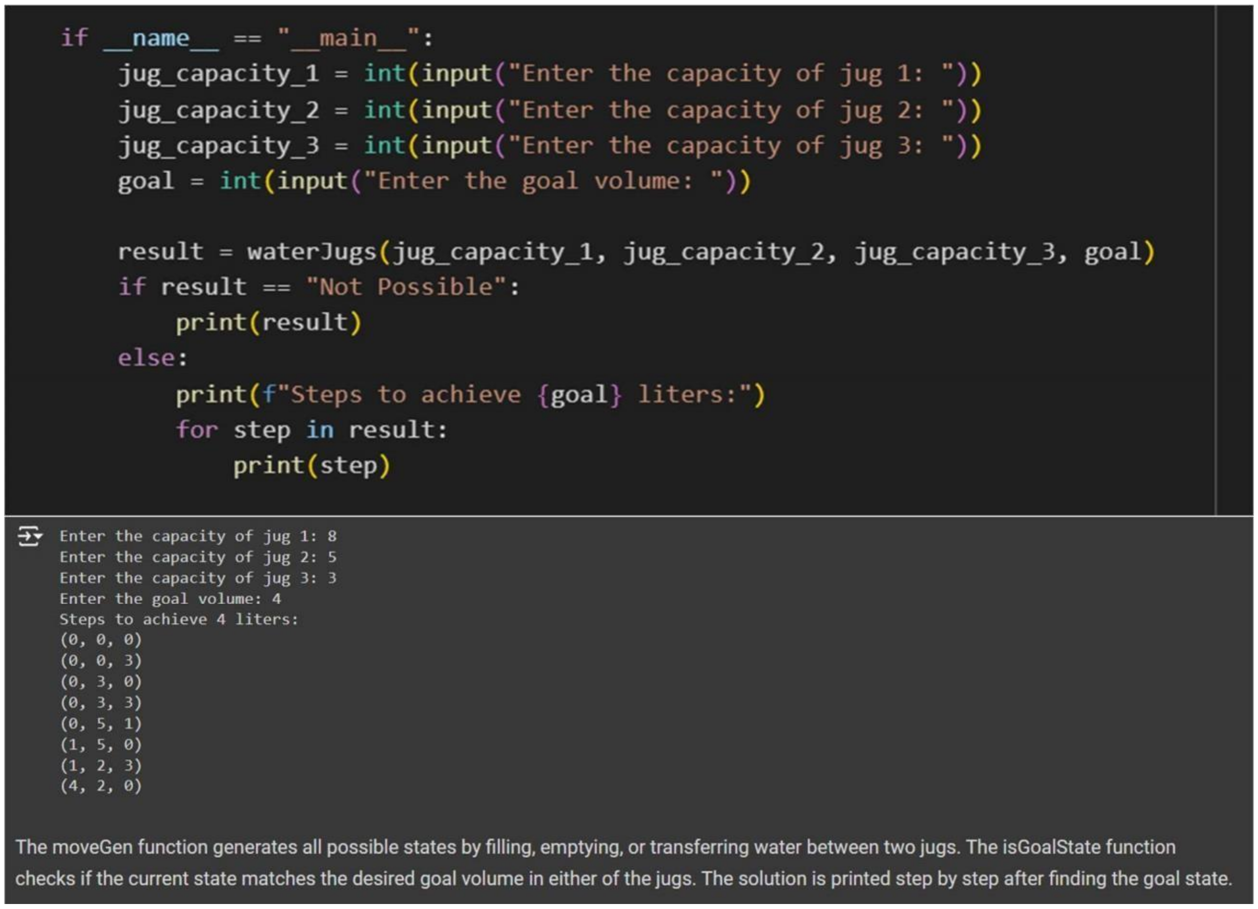
```python
def moveGen(state, jug_capacity_1, jug_capacity_2, jug_capacity_3):
    moves = []
    # Fill Jug 1
    if state[0] < jug_capacity_1:
        moves.append((jug_capacity_1, state[1], state[2]))
    # Fill Jug 2
    if state[1] < jug_capacity_2:
        moves.append((state[0], jug_capacity_2, state[2]))
    # Fill Jug 3
    if state[2] < jug_capacity_3:
        moves.append((state[0], state[1], jug_capacity_3))
    # Empty Jug 1
    if state[0] > 0:
        moves.append((0, state[1], state[2]))
    # Empty Jug 2
    if state[1] > 0:
        moves.append((state[0], 0, state[2]))
    # Empty Jug 3
    if state[2] > 0:
        moves.append((state[0], state[1], 0))
    # Transfer Jug 1 to Jug 2
    transfer = min(state[0], jug_capacity_2 - state[1])
    if transfer > 0:
        moves.append((state[0] - transfer, state[1] + transfer, state[2]))
    # Transfer Jug 1 to Jug 3
    transfer = min(state[0], jug_capacity_3 - state[2])
    if transfer > 0:
        moves.append((state[0] - transfer, state[1], state[2] + transfer))
    # Transfer Jug 2 to Jug 1
    transfer = min(state[1], jug_capacity_1 - state[0])
```



```
    # Transfer Jug 1 to Jug 3
    transfer = min(state[0], jug_capacity_3 - state[2])
    if transfer > 0:
        moves.append((state[0] - transfer, state[1], state[2] + transfer))
    # Transfer Jug 2 to Jug 1
    transfer = min(state[1], jug_capacity_1 - state[0])
    if transfer > 0:
        moves.append((state[0] + transfer, state[1] - transfer, state[2]))
    # Transfer Jug 2 to Jug 3
    transfer = min(state[1], jug_capacity_3 - state[2])
    if transfer > 0:
        moves.append((state[0], state[1] - transfer, state[2] + transfer))
    # Transfer Jug 3 to Jug 1
    transfer = min(state[2], jug_capacity_1 - state[0])
    if transfer > 0:
        moves.append((state[0] + transfer, state[1], state[2] - transfer))
    # Transfer Jug 3 to Jug 2
    transfer = min(state[2], jug_capacity_2 - state[1])
    if transfer > 0:
        moves.append((state[0], state[1] + transfer, state[2] - transfer))
    return moves

def isGoalState(state, goal):
    return state[0] == goal or state[1] == goal or state[2] == goal

def getPath(close, state):
    path = []
    while state:
        path.append(state)
        state = close[state]
```



```python
def getPath(close, state):
    path = []
    while state:
        path.append(state)
        state = close[state]
    return path[::-1]

def waterJugs(jug_capacity_1, jug_capacity_2, jug_capacity_3, goal):
    start_state = (0, 0, 0)
    stack = []
    close = {}
    stack.append(start_state)
    close[start_state] = None
    while stack:
        state = stack.pop()
        if isGoalState(state, goal):
            return getPath(close, state)
        moves = moveGen(state, jug_capacity_1, jug_capacity_2, jug_capacity_3)
        for move in moves:
            if move in close:
                continue
            stack.append(move)
            close[move] = state
    return "Not Possible"

if __name__ == "__main__":
    jug_capacity_1 = int(input("Enter the capacity of jug 1: "))
    jug_capacity_2 = int(input("Enter the capacity of jug 2: "))
    jug_capacity_3 = int(input("Enter the capacity of jug 3: "))
```

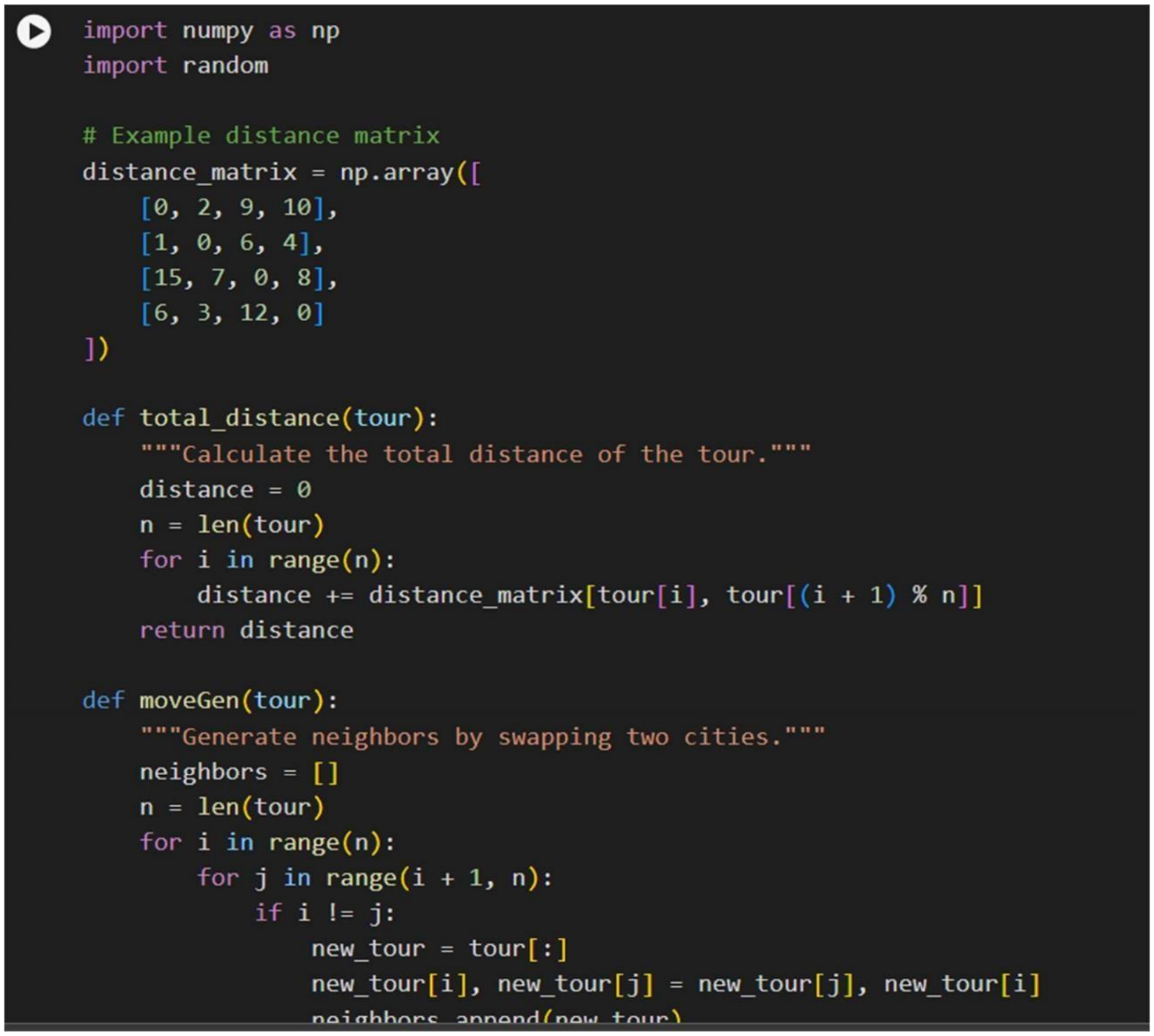


```python
if __name__ == "__main__":
    jug_capacity_1 = int(input("Enter the capacity of jug 1: "))
    jug_capacity_2 = int(input("Enter the capacity of jug 2: "))
    jug_capacity_3 = int(input("Enter the capacity of jug 3: "))
    goal = int(input("Enter the goal volume: "))

    result = waterJugs(jug_capacity_1, jug_capacity_2, jug_capacity_3, goal)
    if result == "Not Possible":
        print(result)
    else:
        print(f"Steps to achieve {goal} liters:")
        for step in result:
            print(step)
```

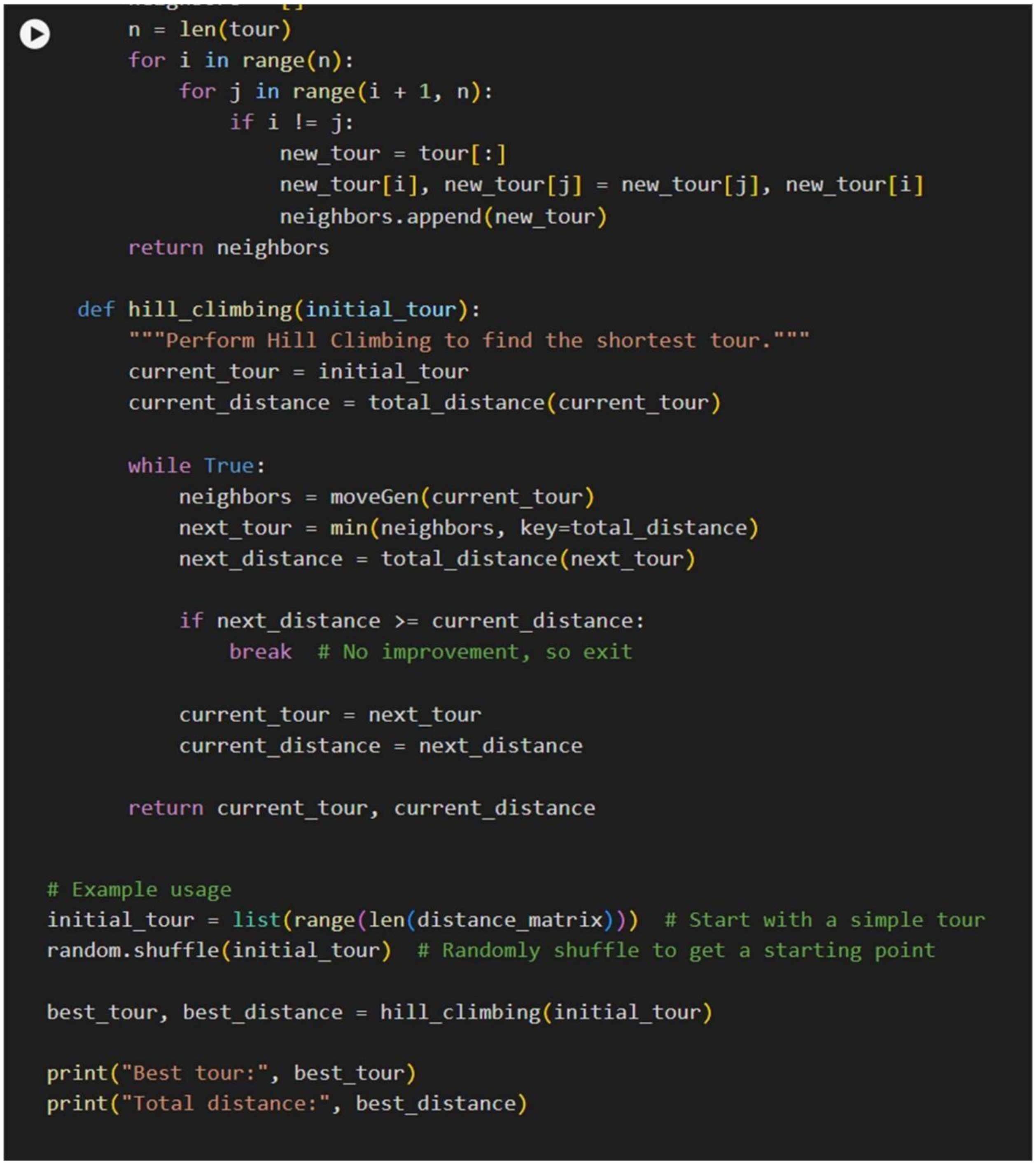
```
Enter the capacity of jug 1: 8
Enter the capacity of jug 2: 5
Enter the capacity of jug 3: 3
Enter the goal volume: 4
Steps to achieve 4 liters:
(0, 0, 0)
(0, 0, 3)
(0, 3, 0)
(0, 3, 3)
(0, 5, 1)
(1, 5, 0)
(1, 2, 3)
(4, 2, 0)
```

The moveGen function generates all possible states by filling, emptying, or transferring water between two jugs. The isGoalState function checks if the current state matches the desired goal volume in either of the jugs. The solution is printed step by step after finding the goal state.

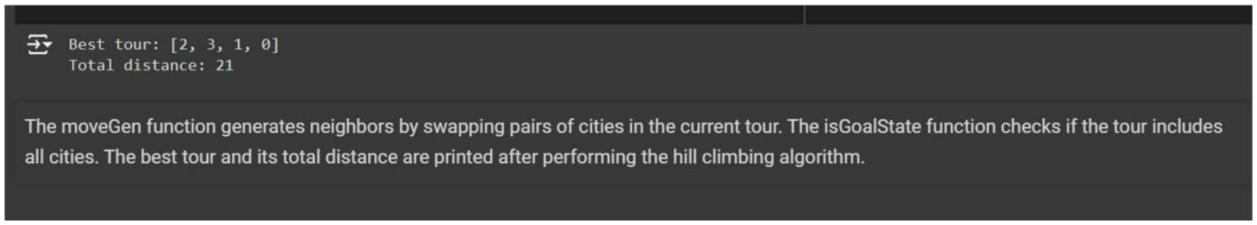
2. Travelling Salesman Problem
A salesman is travelling and selling his/her product to in different cities. The condition is that it has to travel each city just once.



```python
import numpy as np
import random

# Example distance matrix
distance_matrix = np.array([
    [0, 2, 9, 10],
    [1, 0, 6, 4],
    [15, 7, 0, 8],
    [6, 3, 12, 0]
])

def total_distance(tour):
    """Calculate the total distance of the tour."""
    distance = 0
    n = len(tour)
    for i in range(n):
        distance += distance_matrix[tour[i], tour[(i + 1) % n]]
    return distance

def moveGen(tour):
    """Generate neighbors by swapping two cities."""
    neighbors = []
    n = len(tour)
    for i in range(n):
        for j in range(i + 1, n):
            if i != j:
                new_tour = tour[:]
                new_tour[i], new_tour[j] = new_tour[j], new_tour[i]
                neighbors.append(new_tour)
```



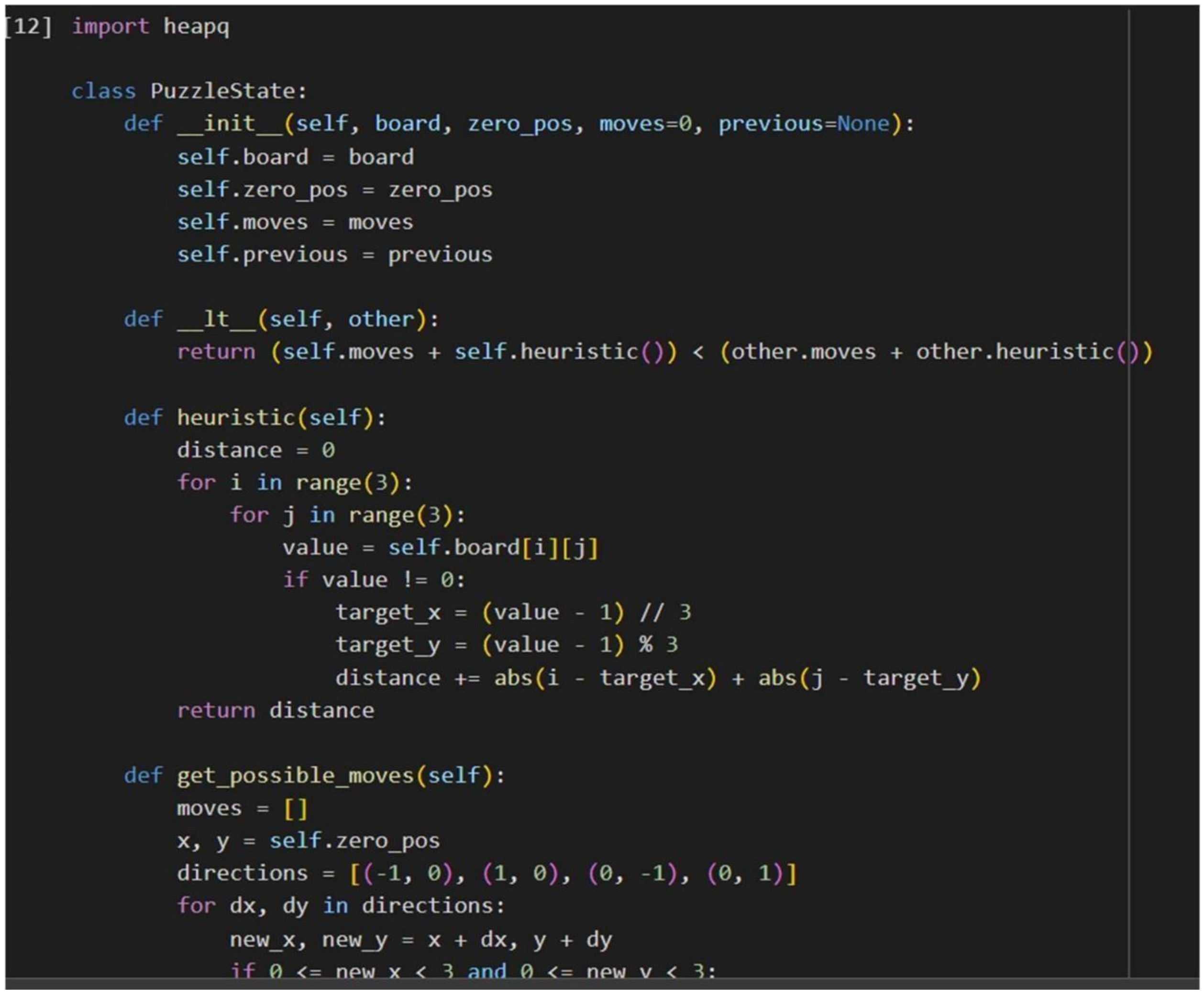
```
      n = len(tour)
      for i in range(n):
          for j in range(i + 1, n):
              if i != j:
                  new_tour = tour[:]
                  new_tour[i], new_tour[j] = new_tour[j], new_tour[i]
                  neighbors.append(new_tour)
      return neighbors

  def hill_climbing(initial_tour):
      """Perform Hill Climbing to find the shortest tour."""
      current_tour = initial_tour
      current_distance = total_distance(current_tour)

      while True:
          neighbors = moveGen(current_tour)
          next_tour = min(neighbors, key=total_distance)
          next_distance = total_distance(next_tour)

          if next_distance >= current_distance:
              break  # No improvement, so exit

          current_tour = next_tour
          current_distance = next_distance

      return current_tour, current_distance


# Example usage
initial_tour = list(range(len(distance_matrix)))  # Start with a simple tour
random.shuffle(initial_tour)  # Randomly shuffle to get a starting point

best_tour, best_distance = hill_climbing(initial_tour)

print("Best tour:", best_tour)
print("Total distance:", best_distance)
```



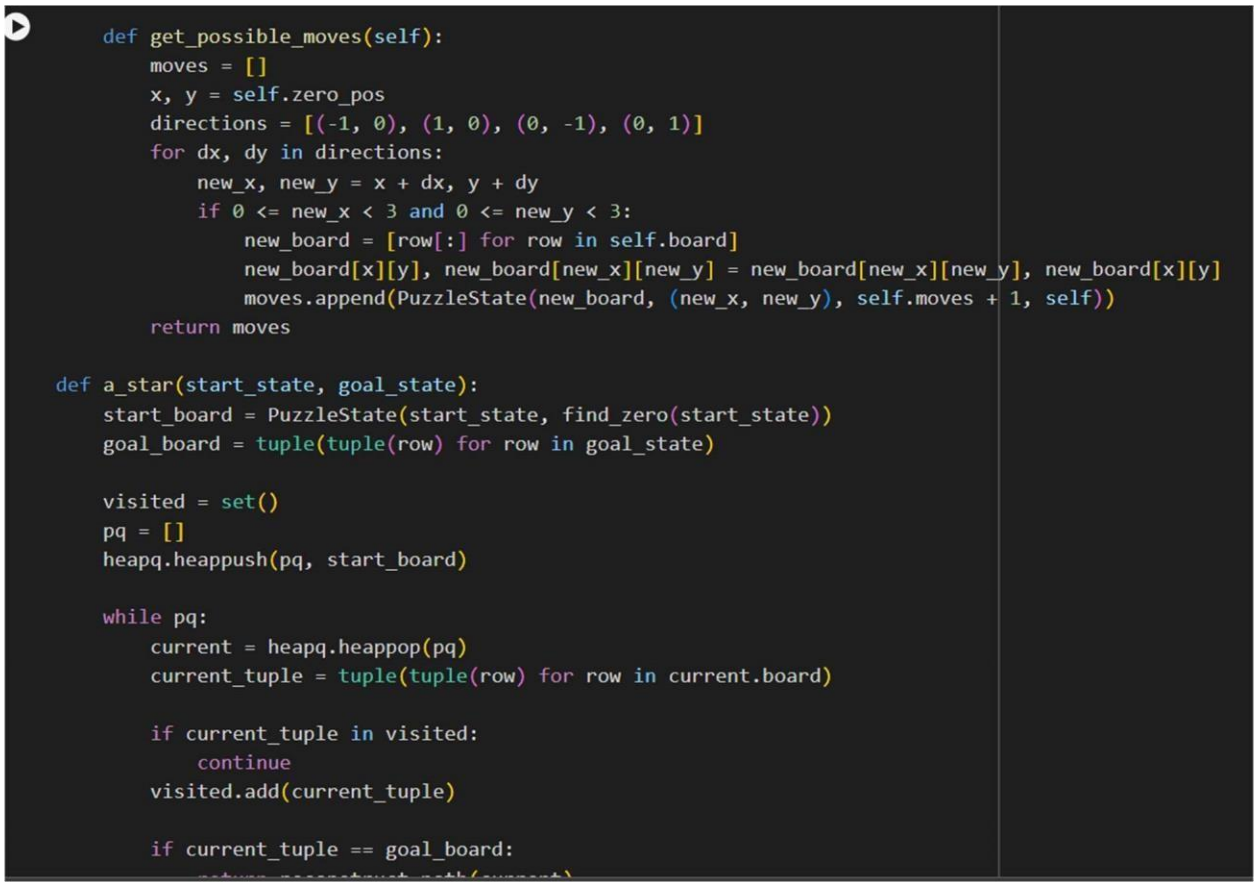
```
Best tour: [2, 3, 1, 0]
Total distance: 21
```

The moveGen function generates neighbors by swapping pairs of cities in the current tour. The isGoalState function checks if the tour includes all cities. The best tour and its total distance are printed after performing the hill climbing algorithm.

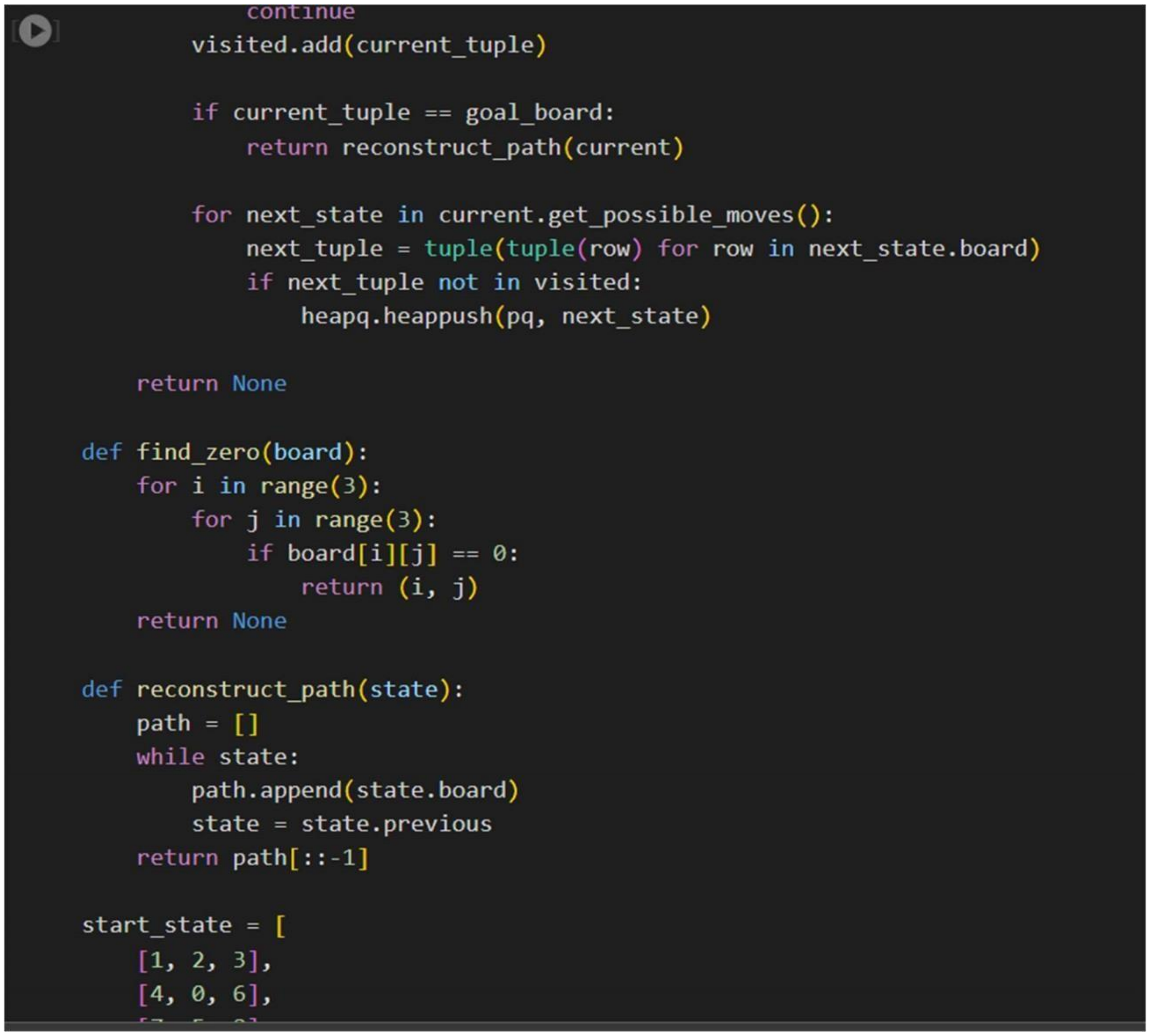
3. 8 Puzzle Problem
An initial state is given in a 8 puzzle where one place is blank out of 9 places. You can shift this blank space and get a different state to reach to a given goal state.



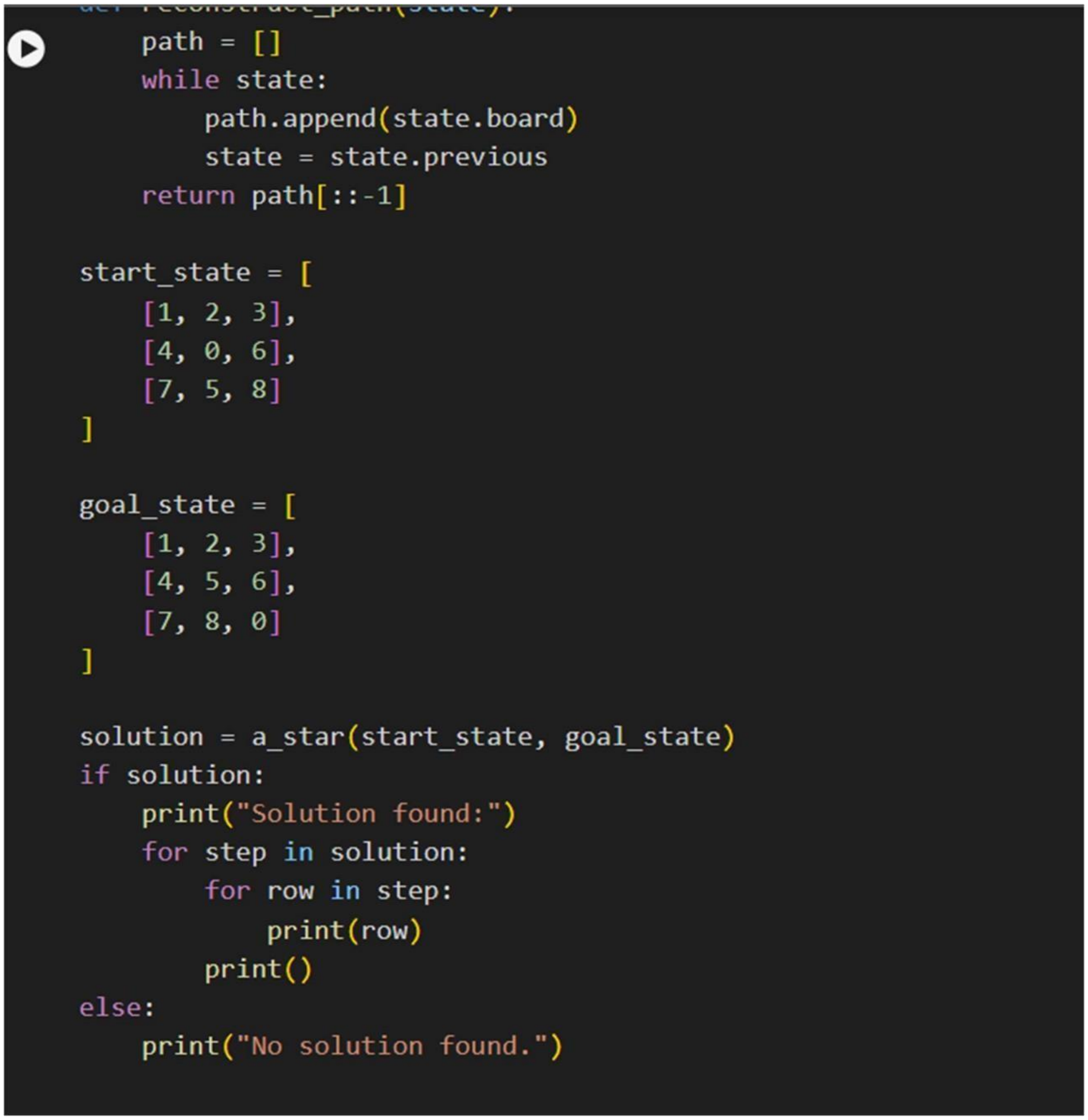
```
[12] import heapq

     class PuzzleState:
         def __init__(self, board, zero_pos, moves=0, previous=None):
             self.board = board
             self.zero_pos = zero_pos
             self.moves = moves
             self.previous = previous

         def __lt__(self, other):
             return (self.moves + self.heuristic()) < (other.moves + other.heuristic())

         def heuristic(self):
             distance = 0
             for i in range(3):
                 for j in range(3):
                     value = self.board[i][j]
                     if value != 0:
                         target_x = (value - 1) // 3
                         target_y = (value - 1) % 3
                         distance += abs(i - target_x) + abs(j - target_y)
             return distance

         def get_possible_moves(self):
             moves = []
             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:
```



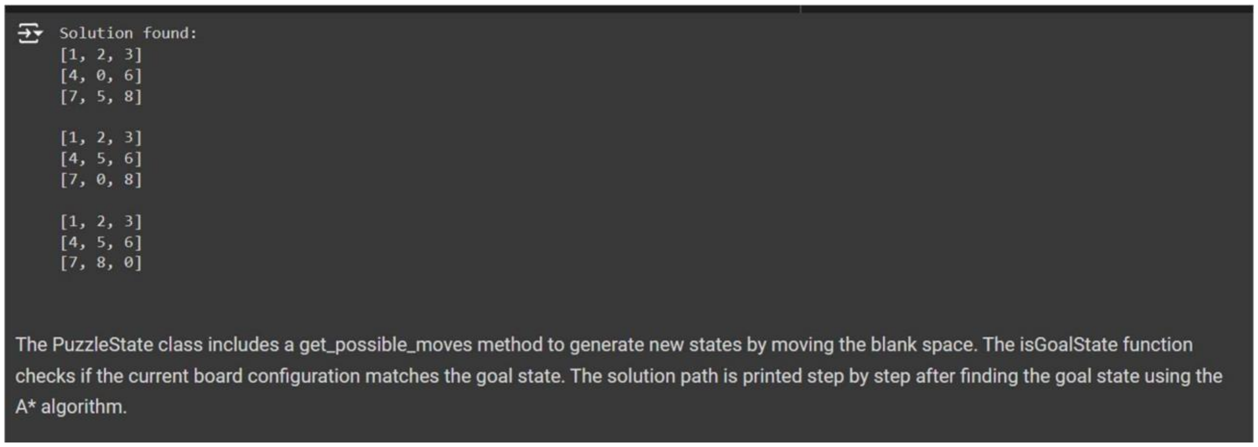
```
    def get_possible_moves(self):
        moves = []
        x, y = self.zero_pos
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] 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]
                moves.append(PuzzleState(new_board, (new_x, new_y), self.moves + 1, self))
        return moves

def a_star(start_state, goal_state):
    start_board = PuzzleState(start_state, find_zero(start_state))
    goal_board = tuple(tuple(row) for row in goal_state)

    visited = set()
    pq = []
    heapq.heappush(pq, start_board)

    while pq:
        current = heapq.heappop(pq)
        current_tuple = tuple(tuple(row) for row in current.board)

        if current_tuple in visited:
            continue
        visited.add(current_tuple)

        if current_tuple == goal_board:
```



```
            continue
        visited.add(current_tuple)

        if current_tuple == goal_board:
            return reconstruct_path(current)

        for next_state in current.get_possible_moves():
            next_tuple = tuple(tuple(row) for row in next_state.board)
            if next_tuple not in visited:
                heapq.heappush(pq, next_state)

    return None

def find_zero(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return (i, j)
    return None

def reconstruct_path(state):
    path = []
    while state:
        path.append(state.board)
        state = state.previous
    return path[::-1]

start_state = [
    [1, 2, 3],
    [4, 0, 6],
```



```
def reconstruct_path(state):
    path = []
    while state:
        path.append(state.board)
        state = state.previous
    return path[::-1]

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

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

solution = a_star(start_state, goal_state)
if solution:
    print("Solution found:")
    for step in solution:
        for row in step:
            print(row)
        print()
else:
    print("No solution found.")
```



```
Solution found:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

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

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

The PuzzleState class includes a get_possible_moves method to generate new states by moving the blank space. The isGoalState function checks if the current board configuration matches the goal state. The solution path is printed step by step after finding the goal state using the A* algorithm.

CONCLUSION:

The provided solutions for the Water Jug Problem, Travelling Salesman Problem (TSP), and 8-Puzzle Problem demonstrate various algorithmic approaches to classic computational problems. The Water Jug Problem uses a depth-first search (DFS) approach to explore all possible states of three jugs and determines the steps needed to achieve a specific water volume. The Travelling Salesman Problem (TSP) employs a hill climbing algorithm to approximate the shortest possible route by iteratively improving the tour through neighbor swaps. The 8-Puzzle Problem utilizes the A* search algorithm, guided by the Manhattan distance heuristic, to find the optimal sequence of moves to reach the goal state. Each solution effectively applies relevant techniques to solve complex problems and print the necessary steps or indicate if a solution is not possible.