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: 2023-24

Experiment 1

(Problem Solving)

Name: Bhuvi Ghosh **Sap ID: 60009210191** **Batch: D22**

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.

```
a = int(input("Enter water level in A: "))
b = int(input("Enter water level in B: "))
c = int(input("Enter water level in C: "))
current_state = (a, b, c)

def move_gen(current):
    moves = []
    a, b, c = current_state
    max_a, max_b, max_c = (3, 5, 8)
    print(f"Current State: A={a}, B={b}, C={c}")
    if a > 0:
        if b < max_b:
            pour= min(a, max_b - b)
            new_state = (a - pour, b + pour, c)
            moves.append(new_state)
        if c < max_c:
            pour= min(a, max_c - c)
            new_state = (a - pour, b, c + pour)
            moves.append(new_state)
    if b > 0:
        if a < max_a:
            pour_amount = min(b, max_a - a)
            new_state = (a + pour_amount, b - pour_amount, c)
            moves.append(new_state)
        if c < max_c:
            pour_amount = min(b, max_c - c)
            new_state = (a, b - pour_amount, c + pour_amount)
            moves.append(new_state)

    if c > 0:
        if a < max_a:
            pour_amount = min(c, max_a - a)
            new_state = (a + pour_amount, b, c - pour_amount)
            moves.append(new_state)
        if b < max_b:
            pour_amount = min(c, max_b - b)
            new_state = (a, b + pour_amount, c - pour_amount)
```

```
            moves.append(new_state)
    return moves
valid_moves = move_gen(current_state)

print("\nValid Moves:")
for move in valid_moves:
    print(move)
```

```
Enter water level in A: 3
Enter water level in B: 5
Enter water level in C: 8
Current State: A=0, B=0, C=8
Valid Moves:
(3, 5, 0)
(0, 5, 3)
(3, 0, 5)
```

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 itertools

def calculate_total_distance(tour, distance_matrix):
    total_distance = 0
    for i in range(len(tour) - 1):
        total_distance += distance_matrix[tour[i]][tour[i + 1]]
    total_distance += distance_matrix[tour[-1]][tour[0]]
    return total_distance

def generate_all_tsp_states(cities, distance_matrix):
    all_states = []
    num_cities = len(cities)
    all_permutations = list(itertools.permutations(range(num_cities)))

    for permutation in all_permutations:
        total_distance = calculate_total_distance(permutation, distance_matrix)
        all_states.append((permutation, total_distance))

    return all_states

if __name__ == "__main__":
    cities = ["City A", "City B", "City C", "City D"]
    distance_matrix = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]

    all_states = generate_all_tsp_states(cities, distance_matrix)

    for state in all_states:
        print(f"Tour: {state[0]}, Total Distance: {state[1]}")
```

```
Tour: (0, 1, 2, 3), Total Distance: 95
Tour: (0, 1, 3, 2), Total Distance: 80
Tour: (0, 2, 1, 3), Total Distance: 95
Tour: (0, 2, 3, 1), Total Distance: 80
Tour: (0, 3, 1, 2), Total Distance: 95
Tour: (0, 3, 2, 1), Total Distance: 95
Tour: (1, 0, 2, 3), Total Distance: 80
Tour: (1, 0, 3, 2), Total Distance: 95
Tour: (1, 2, 0, 3), Total Distance: 95
Tour: (1, 2, 3, 0), Total Distance: 95
Tour: (1, 3, 0, 2), Total Distance: 95
Tour: (1, 3, 2, 0), Total Distance: 80
Tour: (2, 0, 1, 3), Total Distance: 80
Tour: (2, 0, 3, 1), Total Distance: 95
Tour: (2, 1, 0, 3), Total Distance: 95
Tour: (2, 1, 3, 0), Total Distance: 95
Tour: (2, 3, 0, 1), Total Distance: 95
Tour: (2, 3, 1, 0), Total Distance: 80
Tour: (3, 0, 1, 2), Total Distance: 95
Tour: (3, 0, 2, 1), Total Distance: 95
Tour: (3, 1, 0, 2), Total Distance: 80
Tour: (3, 1, 2, 0), Total Distance: 95
Tour: (3, 2, 0, 1), Total Distance: 80
Tour: (3, 2, 1, 0), Total Distance: 95
```

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.

```python
def movegen(board):
  possible_moves = []
  for row in range(3):
    for col in range(3):
      if(board[row][col] == 0):
        empty_row = row
        empty_col = col

  moves = ["Right","Left","Up","Down"]
  for move in moves:
    if move == "Right":
      new_row = empty_row
      new_col = empty_col + 1
    elif move == "Left":
      new_row = empty_row
      new_col = empty_col - 1
    elif move == "Up":
      new_row = empty_row - 1
      new_col = empty_col
    elif move == "Down":
      new_row = empty_row + 1
      new_col = empty_col
    if (0 <= new_row < 3) and (0 <= new_col < 3):
      new_board = [row[:] for row in board]
      new_board[empty_row][empty_col], new_board[new_row][new_col] = new_board[new_row][new_col], new_board[empty_row][empty_col]
      possible_moves.append((f"move - {move}", new_board))

  return possible_moves

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

moves = movegen(initial_board)
for move in moves:
    print(move)
```

```
('move - Right', [[1, 2, 3], [4, 5, 0], [6, 7, 8]])
('move - Left', [[1, 2, 3], [0, 4, 5], [6, 7, 8]])
('move - Up', [[1, 0, 3], [4, 2, 5], [6, 7, 8]])
('move - Down', [[1, 2, 3], [4, 7, 5], [6, 0, 8]])
```