5. Implementation of TSP using heuristic approach.

```
# Traveling Salesman Problem using
# Branch and Bound.
import math
maxsize = float('inf')

# Function to copy temporary solution
# to the final solution
def copyToFinal(curr_path):
	final_path[:N + 1] = curr_path[:]
	final_path[N] = curr_path[0]

# Function to find the minimum edge cost
# having an end at the vertex i
def firstMin(adj, i):
	min = maxsize
	for k in range(N):
		if adj[i][k] < min and i != k:
			min = adj[i][k]
	return min
# function to find the second minimum edge
# cost having an end at the vertex i
def secondMin(adj, i):
	first, second = maxsize, maxsize
	for j in range(N):
		if i == j:
			continue
		if adj[i][j] <= first:
			second = first
			first = adj[i][j]

		elif(adj[i][j] <= second and
			adj[i][j] != first):
			second = adj[i][j]
	return second
# function that takes as arguments:
# curr_bound -> lower bound of the root node
# curr_weight-> stores the weight of the path so far
# level-> current level while moving
# in the search space tree
# curr_path[] -> where the solution is being stored
# which would later be copied to final_path[]
def TSPRec(adj, curr_bound, curr_weight,
			level, curr_path, visited):
	global final_res
```

```
# base case is when we have reached level N
# which means we have covered all the nodes once
if level == N:
    # check if there is an edge from
    # last vertex in path back to the first vertex
    if adj[curr_path[level - 1]][curr_path[0]] != 0:

        # curr_res has the total weight
        # of the solution we got
        curr_res = curr_weight + adj[curr_path[level - 1]]\
                                                [curr_path[0]]
        if curr_res < final_res:
            copyToFinal(curr_path)
            final_res = curr_res
    return
# for any other level iterate for all vertices
# to build the search space tree recursively
for i in range(N):
    # Consider next vertex if it is not same
    # (diagonal entry in adjacency matrix and
    # not visited already)
    if (adj[curr_path[level-1]][i] != 0 and
                            visited[i] == False):
        temp = curr_bound
        curr_weight += adj[curr_path[level - 1]][i]

        # different computation of curr_bound
        # for level 2 from the other levels
        if level == 1:
            curr_bound -= ((firstMin(adj, curr_path[level - 1]) +
                            firstMin(adj, i)) / 2)
        else:
            curr_bound -= ((secondMin(adj, curr_path[level - 1]) +
                            firstMin(adj, i)) / 2)

        # curr_bound + curr_weight is the actual lower bound
        # for the node that we have arrived on.
        # If current lower bound < final_res,
        # we need to explore the node further
        if curr_bound + curr_weight < final_res:
            curr_path[level] = i
            visited[i] = True

            # call TSPRec for the next level
            TSPRec(adj, curr_bound, curr_weight,
```

```
                    level + 1, curr_path, visited)

            # Else we have to prune the node by resetting
            # all changes to curr_weight and curr_bound
            curr_weight -= adj[curr_path[level - 1]][i]
            curr_bound = temp

            # Also reset the visited array
            visited = [False] * len(visited)
            for j in range(level):
                if curr_path[j] != -1:
                    visited[curr_path[j]] = True

# This function sets up final_path
def TSP(adj):
    # Calculate initial lower bound for the root node
    # using the formula 1/2 * (sum of first min +
    # second min) for all edges. Also initialize the
    # curr_path and visited array
    curr_bound = 0
    curr_path = [-1] * (N + 1)
    visited = [False] * N

    # Compute initial bound
    for i in range(N):
        curr_bound += (firstMin(adj, i) +
                    secondMin(adj, i))
    # Rounding off the lower bound to an integer
    curr_bound = math.ceil(curr_bound / 2)
    # We start at vertex 1 so the first vertex
    # in curr_path[] is 0
    visited[0] = True
    curr_path[0] = 0
    # Call to TSPRec for curr_weight
    # equal to 0 and level 1
    TSPRec(adj, curr_bound, 0, 1, curr_path, visited)
# Driver code
# Adjacency matrix for the given graph
adj = [[0, 4, 12, 7],
    [5, 0, 0, 18],
    [11, 0, 0, 6],
    [10, 2, 3, 0]]
N = 4
# final_path[] stores the final solution
# i.e. the // path of the salesman.
final_path = [None] * (N + 1)
```

```
# visited[] keeps track of the already
# visited nodes in a particular path
visited = [False] * N
# Stores the final minimum weight
# of shortest tour.
final_res = maxsize
TSP(adj)
print("Minimum cost :", final_res)
print("Path Taken : ", end = ' ')
for i in range(N + 1):
	print(final_path[i], end = ' ')
```

Output:

Minimum cost : 25

Path Taken : 0 2 3 1 0