EXPERIMENT NO. 5

Student Name and Roll Number: Piyush Gambhir – 21CSU349	
sSemester /Section: Semester-V – AIML-V-B (AL-3)	
Link to Code: NCU-Lab-Manual-And-End-Semester-Projects/NCU-CSL347-AAIES-Lab_Manual at main · Piyush-Gambhir/NCU-Lab-Manual-And-End-Semester-Projects (github.com)	
Date:	
Faculty Signature:	
Grade:	

Objective(s):

- Understand what Branch and Bound is.
- Study about different algorithm design paradigms.
- Implement Branch and Bound for solving a real-world graph-based problem.

Outcome:

Students will be familiarized with application of Branch and Bound Algorithm on graph-based problems.

Problem Statement:

Write a Python program to solve the Travelling Salesman problem using Branch and Bound approach.

Background Study:

Branch and Bound is a systematic algorithmic technique used to solve optimization problems by exploring the solution space efficiently. It partitions the search space into smaller subproblems, or branches, and establishes upper and lower bounds on their potential solutions. By pruning branches with solutions that cannot possibly improve the current best-known solution, it reduces the search space, leading to faster convergence towards the optimal solution

Question Bank:

1. What is Branch and Bound Approach?
   The Branch and Bound approach is an optimization technique used to solve combinatorial optimization problems. It systematically explores the solution space by dividing it into smaller subproblems or "branches" and then prunes certain branches based on upper and lower bounds. This method is commonly applied to problems like the Traveling Salesman Problem and knapsack problems, helping to find the optimal or near-optimal solutions efficiently.

2. What type of problem is travelling salesman?
   The Traveling Salesman Problem is a classic optimization problem in which a salesperson aims to find the shortest possible route that visits a set of cities exactly once and returns to the starting city. It's a well-known NP-hard problem, meaning that as the number of cities increases, finding the exact optimal solution becomes exponentially more difficult. The TSP has applications in various fields such as logistics, transportation, and manufacturing.

Student Work Area

Algorithm/Flowchart/Code/Sample Outputs

Code:

Experiment 5

Problem Statement

Write a Python program to solve the Travelling Salesman problem using Branch and Bound approach.

Imagine a salesman who needs to visit a set of cities and return to his starting point while minimizing the total distance traveled. Let's consider a small set of cities with their pairwise distances:

- City A to City B: 10 miles
- City A to City C: 15 miles
- City A to City D: 20 miles
- City B to City C: 35 miles
- City B to City D: 25 miles
- City C to City D: 30 miles

The goal of the TSP is to find the shortest possible route that visits each city exactly once and returns to the starting city.

Expectation From The Code

1. Cost Matrix
2. Reduced cost matrix
3. All the intermediate matrices (reduced cost) formed during the process to find cost of a path
4. And finally the cost

Code:

```
import math

# Global Variables
infinity = float('inf')  # Represents an unbounded upper value for comparison
num_nodes = 4  # Total number of nodes in the graph

# Variables to store the result
final_path = [None] * (num_nodes + 1)
final_min_cost = infinity


# Function to update the final_path array
def updateFinalPath(curr_path):
    global num_nodes, final_path
    final_path[:num_nodes + 1] = curr_path[:]
    final_path[num_nodes] = curr_path[0]


# Function to find the minimum edge cost from a given node
def getFirstMinCost(adj_matrix, index):
    return min(adj_matrix[index][j] for j in range(num_nodes) if index != j)


```

```python
# Function to find the second minimum edge cost from a given node
def getSecondMinCost(adj_matrix, index):
    vals = [adj_matrix[index][j] for j in range(num_nodes) if index != j]
    first, second = sorted(vals)[:2]
    return second


# Recursive function to solve the TSP problem
def TSPRecursive(adj_matrix, curr_bound, curr_cost, level, curr_path, visited_nodes):
    global final_min_cost, num_nodes

    # base case: if we have reached the last node and there is an edge
    # from the last node to the first node
    if level == num_nodes:
        if adj_matrix[curr_path[level - 1]][curr_path[0]] != 0:
            curr_total_cost = curr_cost + \
                adj_matrix[curr_path[level - 1]][curr_path[0]]
            if curr_total_cost < final_min_cost:
                updateFinalPath(curr_path)
                final_min_cost = curr_total_cost
        return

    # Loop through all vertices and recurse
    for i in range(num_nodes):
        if adj_matrix[curr_path[level - 1]][i] != 0 and visited_nodes[i] == False:
            temp_bound = curr_bound
```

```python
            curr_cost += adj_matrix[curr_path[level - 1]][i]

            # Calculate a new lower bound
            if level == 1:
                curr_bound -= ((getFirstMinCost(adj_matrix, curr_path[level - 1]) +
                                getFirstMinCost(adj_matrix, i)) / 2)
            else:
                curr_bound -= ((getSecondMinCost(adj_matrix, curr_path[level - 1]) +
                                getFirstMinCost(adj_matrix, i)) / 2)

            # If the new lower bound + current cost is less than final_min_cost,
            # continue with this path
            if curr_bound + curr_cost < final_min_cost:
                curr_path[level] = i
                visited_nodes[i] = True

                TSPRecursive(adj_matrix, curr_bound, curr_cost,
                             level + 1, curr_path, visited_nodes)

            # Reset variables for next iteration
            curr_cost -= adj_matrix[curr_path[level - 1]][i]
            curr_bound = temp_bound
            visited_nodes[i] = False

```

```python
def TSP(adj_matrix):
    global final_min_cost, num_nodes

    # Initialize variables for TSP
    curr_bound = 0
    curr_path = [-1] * (num_nodes + 1)
    visited_nodes = [False] * num_nodes

    # Calculate initial lower bound
    for i in range(num_nodes):
        curr_bound += (getFirstMinCost(adj_matrix, i) +
                       getSecondMinCost(adj_matrix, i))
    curr_bound = math.ceil(curr_bound / 2)

    # Start from vertex 0
    visited_nodes[0] = True
    curr_path[0] = 0

    # Call recursive TSP function
    TSPRecursive(adj_matrix, curr_bound, 0, 1, curr_path, visited_nodes)

    # Print the final result
    print("\nMinimum cost:", final_min_cost)
    print("Path Taken:", ' '.join(map(str, final_path)))

```

Output:

```python
# Example adjacency matrix
adj_matrix = [[0, 10, 15, 20],
              [10, 0, 35, 25],
              [15, 35, 0, 30],
              [20, 25, 30, 0]]

print("Cost Matrix: ")
for row in adj_matrix:
    print(row)


TSP(adj_matrix)
```

[11] ✓ 0.0s Python

```
Cost Matrix:
[0, 10, 15, 20]
[10, 0, 35, 25]
[15, 35, 0, 30]
[20, 25, 30, 0]

Minimum cost: 80
Path Taken: 0 1 3 2 0
```