	Pimpri Chinchwad Education Trust's Pimpri Chinchwad College of Engineering
Assignment No: 8	

Assignment: Optimizing Delivery Routes for a Logistics Company (TSP — Least Cost Branch & Bound)

1. Problem Statement

A leading logistics company, SwiftShip, is responsible for delivering packages to multiple cities. To minimize fuel costs and delivery time, the company needs to find the shortest possible route that allows a delivery truck to visit each city exactly once and return to the starting point. The company wants an optimized solution that guarantees the least cost route, considering:

- Varying distances between cities.
- Fuel consumption costs, which depend on road conditions.
- Time constraints, as deliveries must be completed within a given period. Since there are N cities, a brute-force approach checking all (N-1)!permutations is infeasible for large N (e.g., 20+ cities). Therefore, you must implement an LC (Least Cost) Branch and Bound algorithm to find the optimal route while reducing unnecessary computations efficiently.

2. Course Objective

1. To know the basics of computational complexity of various algorithms.
2. To select appropriate algorithm design strategies to solve real-world problems.

3. Course Outcomes

1. Apply branch & Bound technique to solve problems

1. Theory / Background

- The problem is the Travelling Salesman Problem (TSP): find minimum-cost cycle visiting each city once.
- Branch & Bound explores partial solutions (paths) as nodes in a search tree. Each node has:
  - a partial path,
  - current cost so far,
  - a lower bound estimate on completion cost (computed via matrix reduction).
- We expand nodes in increasing lower-bound order (best-first). If a node's lower bound ≥ current best solution cost, prune it.
- Matrix reduction: set impossible edges to ∞, then for each row subtract row minimum, for each column subtract column minimum. The sum of subtractions is a lower bound contribution.
- LC Branch & Bound with reduced matrix is exact and often prunes massively compared to brute-force.

2. Algorithm (LC Branch & Bound — Summary)

1. Build initial cost matrix of size N×N (cost[i][j] = fuel-cost from i→j; diagonal = ∞).
2. Reduce matrix — compute initial lower bound lb0.
3. Create root node with reduced matrix, lb=lb0, path = [startCity], level = 0.
4. Use a priority queue (min-heap) keyed by node.lb. Pop the node with smallest lb.
5. If node.level == N−1 (one city left), finalize path and update best tour + cost.
6. Else for each feasible city j not in path:
   - Create child: copy parent's matrix, set row of parent city to ∞, set column j to ∞, set [j][start] to ∞ when closing final state accordingly, set edge parent→j as used (cost already accounted).
   - Reduce child's matrix to compute new lower bound: child.lb = parent.lb + cost[parent→j] + reductionCost.
   - If child.lb < bestCost, push child to PQ; else prune.
7. Continue until PQ empty.
8. Best solution at termination is optimal route.

3. Pseudocode

```
procedure TSP_BranchAndBound(costMatrix, start):
    rootMatrix, lb0 = reduceMatrix(costMatrix)
    root = Node(matrix=rootMatrix, lb=lb0, path=[start], level=0)
    bestCost = +INF
    bestPath = null
    PQ = min-heap ordered by lb
    PQ.push(root)
    while PQ not empty:
        node = PQ.pop()
        if node.lb >= bestCost: continue // prune
        if node.level == N-1:
            // complete tour: add last city -> start cost
            last = node.path.back()
            totalCost = node.lb + originalCost[last][start]
            if totalCost < bestCost:
                bestCost = totalCost; bestPath = node.path + [start]
            continue
        u = node.path.back()
        for each v not in node.path:
            childMatrix = copy(node.matrix)
            // block row u and column v and set [v][u] = INF (prevent immediately returning)
            setRowToInf(childMatrix, u)
            setColToInf(childMatrix, v)
            childCost = node.lb + originalCost[u][v]
            reductionCost = reduceMatrixInPlace(childMatrix)
            child.lb = childCost + reductionCost
            child.path = node.path + [v]
            child.level = node.level + 1
            if child.lb < bestCost: PQ.push(child)
    return bestCost, bestPath
```

4. Complexity & Notes

- Worst-case time still factorial, but bounding + reduced cost often prunes large parts of tree.
- Memory: PQ can grow large; used memory depends on pruning effectiveness.
- Practical for exact solution up to ~12–16 cities comfortably; higher N may be possible with strong pruning or problem structure.
- Handling time constraints: one can modify bounding to add penalties for late arrival or disallow paths that exceed time windows.

5. Implementation

```
import java.util.*;

public class TSPBranchAndBound {

    static class Node implements Comparable<Node> {
        int level;      // Level in the search tree (number of cities visited)
        int pathCost;   // Cost accumulated so far
        int bound;      // Lower bound of cost to complete the tour
        List<Integer> path; // Cities visited so far

        Node(int level, int pathCost, List<Integer> path) {
            this.level = level;
            this.pathCost = pathCost;
            this.path = new ArrayList<>(path);
        }

        @Override
        public int compareTo(Node o) {
            return this.bound - o.bound; // Min-heap based on bound
        }
    }

    // Compute lower bound for a node
    static int calculateBound(Node node, int[][] costMatrix, int N) {
        int bound = node.pathCost;
        boolean[] visited = new boolean[N];
        for (int city : node.path) visited[city] = true;

        // Add minimum outgoing edge for unvisited cities
        for (int i = 0; i < N; i++) {
            if (!visited[i]) {
                int min = Integer.MAX_VALUE;
                for (int j = 0; j < N; j++) {
                    if (i != j && !visited[j] && costMatrix[i][j] < min) {
                        min = costMatrix[i][j];
                    }
                }
                // If all remaining cities are visited, pick minimum edge to any city
                if (min == Integer.MAX_VALUE) {
                    for (int j = 0; j < N; j++) {
                        if (i != j && costMatrix[i][j] < min) min = costMatrix[i][j];
                    }
```

```
            }
            bound += min;
        }
    }
    return bound;
}

static void tspLCBB(int[][] costMatrix) {
    int N = costMatrix.length;
    PriorityQueue<Node> pq = new PriorityQueue<>();
    List<Integer> path0 = new ArrayList<>();
    path0.add(0); // Start from city 0
    Node root = new Node(0, 0, path0);
    root.bound = calculateBound(root, costMatrix, N);
    pq.add(root);

    int minCost = Integer.MAX_VALUE;
    List<Integer> bestPath = null;

    while (!pq.isEmpty()) {
        Node curr = pq.poll();

        if (curr.bound >= minCost) continue; // Prune

        if (curr.level == N - 1) {
            // Complete tour by returning to start
            int lastCity = curr.path.get(curr.path.size() - 1);
            int totalCost = curr.pathCost + costMatrix[lastCity][0];
            if (totalCost < minCost) {
                minCost = totalCost;
                bestPath = new ArrayList<>(curr.path);
                bestPath.add(0);
            }
            continue;
        }

        int lastCity = curr.path.get(curr.path.size() - 1);
        for (int nextCity = 0; nextCity < N; nextCity++) {
            if (!curr.path.contains(nextCity)) {
                List<Integer> newPath = new ArrayList<>(curr.path);
                newPath.add(nextCity);
                int newCost = curr.pathCost + costMatrix[lastCity][nextCity];
                Node child = new Node(curr.level + 1, newCost, newPath);
                child.bound = calculateBound(child, costMatrix, N);
                if (child.bound < minCost) pq.add(child);
            }
        }
    }

    // Print solution
    System.out.println("\nOptimal TSP route:");
    for (int i = 0; i < bestPath.size(); i++) {
        if (i > 0) System.out.print(" -> ");
        System.out.print(bestPath.get(i));
    }
    System.out.println("\nTotal minimum cost: " + minCost);
```

```
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of cities: ");
        int N = sc.nextInt();
        int[][] costMatrix = new int[N][N];

        System.out.println("Enter cost/distance matrix row by row:");
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                costMatrix[i][j] = sc.nextInt();

        tspLCBB(costMatrix);
        sc.close();
    }
}
```

6. Output

```
Enter number of cities: 4
Enter cost/distance matrix row by row:
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0

Optimal TSP route:
0 -> 1 -> 3 -> 2 -> 0
Total minimum cost: 80
```

7. **Complexity Analysis**

- Best case: pruning is so powerful we examine very few nodes.
- Worst case: behaves like factorial time O(N!) (no pruning possible).
- Bounding (matrix-reduction) cost per node: O(N^2) to copy & reduce matrix.
- Practical observation: Branch & Bound can solve N≈10–15 exactly within seconds; for larger N, runtime grows quickly.

8. **Conclusion**

LC Branch & Bound with reduced cost matrix gives an exact TSP algorithm that drastically reduces the search space compared to naive permutation enumeration by using strong lower bounds and best-first expansion. It is suitable for SwiftShip when the number of stops per route is moderate and exact optimality is required (e.g., high-cost deliveries or regulatory constraints). For larger route sets or real-time routing, combine exact BnB for critical subroutes with heuristics (Christofides, 2-opt, or metaheuristics) for scalability.