	Pimpri Chinchwad Education Trust's **Pimpri Chinchwad College of Engineering**
Assignment 8	

Assignment 08: Optimizing Logistics with Branch & Bound (TSP)

Aim

To design and implement a solution for the Travelling Salesman Problem (TSP) using a Least Cost (LC) Branch and Bound algorithm. The goal is to find the guaranteed shortest possible route for a delivery truck that must visit each city exactly once and return to the starting city, while efficiently pruning the search space to avoid the infeasible $O(N!)$ brute-force approach.

Course Context

- **Course Objectives:**
  - To understand the basics of computational complexity, especially for NP-hard problems.
  - To select and apply appropriate advanced algorithm design strategies, like Branch and Bound, to solve complex real-world problems.
- **Course Outcomes:**
  - Apply the Branch & Bound technique to find optimal solutions for optimization problems.

Concepts

1. The Travelling Salesman Problem (TSP)

The Travelling Salesman Problem is a classic optimization problem. Given a list of cities and the distances (or costs) between each pair, the goal is to find the shortest possible route that visits each city exactly once and returns to the origin city.

- **Graph Model:** The problem is modeled as a complete, weighted graph.
  - **Vertices (V):** Represent the cities.
  - **Edges (E):** Represent the paths between cities.
  - **Weights (W):** Represent the cost or distance of traveling between two cities.
- **The Challenge (NP-Hard):** The brute-force method involves checking every possible permutation of cities. For N cities, this is $(N-1)!$ possible tours. This factorial complexity makes it computationally impossible for even a small number of cities (e.g., 20! is a number with 19 digits).

2. Branch and Bound (B&B)

Branch and Bound is an algorithm design paradigm for solving optimization problems. It systematically explores a

state space tree (the "Branch" part) while using a **lower bound** calculation to discard large parts of the tree that cannot possibly contain the optimal solution (the "Bound" part).

- **State Space Tree:** A tree representing all possible partial solutions.
  - The **root** is the starting city (partial path of length 1).
  - **Children** of a node represent the next possible city to visit.
  - A **leaf** at level N represents a complete tour.
- **Least Cost (LC) B&B:** This is a specific B&B strategy. We use a **min-priority queue** to explore the state space. The priority is given to the node with the **"Least Cost" lower bound**. This ensures we are always exploring the *most promising* partial path first.

3. The Bounding Function: Reduced Cost Matrix

The efficacy of B&B for TSP comes from its bounding function. We must calculate a **lower bound (LB)** for any partial path—a guaranteed minimum cost for *any* full tour that can be built from that partial path.

This is done using a **Reduced Cost Matrix**:

1. **Row Reduction:**
   - For each row in the cost matrix, find the smallest element.
   - Subtract this minimum value from *every* element in that row.
   - Add this minimum value to the total Lower Bound.
   - *Intuition:* The cost of *any* tour *must* include at least one path *from* every city. We are factoring out this minimum cost.
2. **Column Reduction:**
   - After all rows are reduced, repeat the process for each column.
   - Find the minimum value in each column, subtract it from all elements in that column, and add the minimum to the Lower Bound.
   - *Intuition:* Any tour *must* also include at least one path *to* every city.

The sum of all values subtracted during row and column reduction gives a strong, non-zero Lower Bound for the cost of any tour.

4. The LCBB Algorithm for TSP

1. **Initialization:**
   - Create a min-priority queue `pq` to store "live" (in-progress) nodes.
   - Initialize `bestCost = INF`.
   - Create a **root node**:
     - Level = 0, Path = {Start City}.
     - Calculate its `lb` by performing full row and column reduction on the *initial* cost matrix.
   - Push the root node onto `pq`.
2. **Main Loop:**
   - While `pq` is not empty:
     - Pop the node `u` with the *smallest* `lb` from `pq`.
     - **Pruning Step:** If `u.lb >= bestCost`, this path is already worse than a solution we've found. Discard (prune) it and continue the loop.
3. **Goal Check:**
   - If `u` represents a full tour (e.g., `level == N-1`):

        - A complete tour is found. Add the cost of returning to the start city.
        - If this tour's total cost is less than `bestCost`, update `bestCost` and `bestPath`.
        - Continue the loop, as other nodes in the priority queue might still lead to an optimal solution (or this one).
4. **Branching (If not a goal):**
    - For each city `v` that is *not* already in `u.path`:
        - Create a **child node** `c` to represent the path `u -> v`.
        - **Child's Cost:** `c.cost = u.cost + matrix[u.city][v]`.
        - **Child's Matrix:** Copy `u.mat`. To prevent cycles and invalid paths:
            - Set row `u.city` to `INF`.
            - Set col `v` to `INF`.
            - Set `mat[v][start_city]` to `INF` (prevents visiting the start city too early).
        - **Child's Lower Bound:** `c.lb = c.cost + reduce_matrix(c.mat)`. (The `reduce_matrix` function performs row/column reduction on the child's new matrix and returns the *additional* reduction cost).
        - **Push to PQ:** If `c.lb < bestCost`, push `c` onto `pq`.

This "best-first" search ensures that we explore the most promising paths first, allowing for effective pruning.

C++ Implementation

```
#include <iostream>

#include <vector>

#include <queue>      // For priority_queue

#include <limits>     // For INT_MAX

#include <algorithm>  // For fill, find

#include <numeric>    // For iota


using namespace std;


const int INF = numeric_limits<int>::max();

int N; // Number of cities


// Node represents a state in the state space tree
```

```
struct Node {
    vector<vector<int>> mat; // Reduced cost matrix for this node
    vector<int> path;        // Path taken so far
    int cost;                // Cost of the path so far
    int lb;                  // Lower bound cost for this node
    int level;               // Number of cities visited (level in the tree)
    int curr_city;           // The current city this node is at
};

// Custom comparator for the min-priority queue
struct CompareNode {
    bool operator()(const Node& a, const Node& b) {
        return a.lb > b.lb;
    }
};

// Performs row and column reduction on the matrix
// Modifies the matrix in-place and returns the total reduction cost
int reduce_matrix(vector<vector<int>>& mat) {
    int reduction = 0;

    // 1. Row Reduction
    for (int i = 0; i < N; ++i) {
        int min_val = INF;
        for (int j = 0; j < N; ++j) {
            if (mat[i][j] < min_val) {
                min_val = mat[i][j];
            }
        }

        if (min_val != 0 && min_val != INF) {
            for (int j = 0; j < N; ++j) {
                if (mat[i][j] != INF) {
                    mat[i][j] -= min_val;
                }
            }
            reduction += min_val;
        }
    }

    // 2. Column Reduction
    for (int j = 0; j < N; ++j) {
        int min_val = INF;
        for (int i = 0; i < N; ++i) {
            if (mat[i][j] < min_val) {
                min_val = mat[i][j];
            }
        }

        if (min_val != 0 && min_val != INF) {
            for (int i = 0; i < N; ++i) {
                if (mat[i][j] != INF) {
                    mat[i][j] -= min_val;
                }
            }
            reduction += min_val;
        }
    }

    return reduction;
}

// Helper function to check if a city is already in the path
bool isVisited(const vector<int>& path, int city) {
    for (int p : path) {
        if (p == city) return true;
    }
    return false;
}

int main() {
    cout << "Enter the number of cities (N): ";
    cin >> N;

    vector<vector<int>> start_mat(N, vector<int>(N));
    cout << "Enter the cost matrix (use -1 or a large number for INF):\n";
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            cin >> start_mat[i][j];
            if (start_mat[i][j] == -1) {
                start_mat[i][j] = INF;
        }
    }
}

// Priority queue to store live nodes (best-first)
priority_queue<Node, vector<Node>, CompareNode> pq;

// --- Create the Root Node ---
Node root;
root.mat = start_mat;
root.path.push_back(0); // Start at city 0
root.cost = 0;
root.level = 0;
root.curr_city = 0;
root.lb = reduce_matrix(root.mat); // Initial lower bound

pq.push(root);

int bestCost = INF;
vector<int> bestPath;

// --- Main LC Branch & Bound Loop ---
while (!pq.empty()) {
    Node node = pq.top();
    pq.pop();

    // --- Pruning Step ---
// If this node's LB is already worse than our best, prune it.
if (node.lb >= bestCost) {
    continue;
}

// --- Goal Check ---
// If we have visited N cities (level N-1)
if (node.level == N - 1) {
    // This is a complete tour.
    // Add cost of returning to start city.
    if (node.mat[node.curr_city][0] != INF) {
        int final_cost = node.cost + node.mat[node.curr_city][0];

        if (final_cost < bestCost) {
            bestCost = final_cost;
            bestPath = node.path;
            bestPath.push_back(0); // Add return to start
        }
    }
    // Since this is a leaf, we don't branch further.
    continue;
}

// --- Branching Step ---
// For each possible next city 'v'
for (int v = 0; v < N; ++v) {
    // Check if city 'v' is not visited and there is a path
```

```
if (!isVisited(node.path, v) && node.mat[node.curr_city][v] != INF) {

    // Create the child node for path u -> v
    Node child;
    child.mat = node.mat;
    child.path = node.path;
    child.path.push_back(v);
    child.level = node.level + 1;
    child.curr_city = v;

    int u = node.curr_city; // Parent city

    // Cost of path so far
    child.cost = node.cost + node.mat[u][v];

    // --- Prepare Child's Reduced Matrix ---
    // Set row 'u', col 'v', and 'v->start' to INF
    for (int j = 0; j < N; ++j) {
        child.mat[u][j] = INF; // Set row u to INF
    }
    for (int i = 0; i < N; ++i) {
        child.mat[i][v] = INF; // Set col v to INF
    }
    child.mat[v][0] = INF; // Set v -> start_city to INF

    // Get new reduction cost
    int reduction = reduce_matrix(child.mat);
```

```
            // --- Calculate Child's Lower Bound ---
            // Cost so far + LB of remaining problem
            child.lb = child.cost + reduction;

            // --- Push Child if Promising ---
            if (child.lb < bestCost) {
                pq.push(child);
            }
        }
    }
}

// --- Final Output ---
if (bestCost == INF) {
    cout << "No feasible tour found." << endl;
} else {
    cout << "Minimum Cost: " << bestCost << endl;
    cout << "Optimal Path: ";
    for (size_t i = 0; i < bestPath.size(); ++i) {
        cout << bestPath[i];
        if (i < bestPath.size() - 1) cout << " -> ";
    }
    cout << endl;
}

return 0;
```

}

Example Input and Output

Example Input

Enter the number of cities (N): 4

Enter the cost matrix (use -1 or a large number for INF):

-1 10 15 20

10 -1 35 25

15 35 -1 30

20 25 30 -1

Output

Minimum Cost: 80

Optimal Path: 0 -> 1 -> 3 -> 2 -> 0

Note: The path `0 -> 2 -> 3 -> 1 -> 0` also has a cost of 80 and is an equally valid optimal solution.

Complexity Analysis

- **Cost per Node:** To calculate the Lower Bound, we must perform row and column reduction, which takes $O(N^2)$ time. Copying the matrix for a child node also takes $O(N^2)$.
- **Number of Nodes (Worst Case):** In the worst case, we might not prune effectively and end up exploring a factorial number of nodes, similar to brute force. The total worst-case complexity is $O(N^2 \times N!)$.
- **Practical Performance:** In practice, the B&B pruning is very effective. It explores only a tiny fraction of the state space, allowing it to solve "small" instances (e.g., $N=15-20$) exactly, which is impossible for brute-force.

Conclusion

The Least Cost Branch & Bound (LCBB) algorithm is an exact and powerful method for solving NP-hard optimization problems like the TSP. By intelligently exploring the state space (Branching) and using a strong cost-reduction heuristic to prune unpromising paths (Bounding), it can find the guaranteed optimal solution far more efficiently than a naive brute-force approach. This makes it a feasible strategy for real-world logistics problems where finding the optimal route is critical.