Koneru Lakshmaiah Education Foundation
(Deemed to be University estd. u/s. 3 of the UGC Act, 1956)
Off-Campus: Bachupally-Gandimaisamma Road, Bowrampet, Hyderabad, Telangana - 500 043.
Phone No: 7815926816, www.klh.edu.in

Department of Computer Science and Engineering

2025-2026
Odd Semester

DESIGN AND ANALYSIS ALGORITHMS
(24CS2203)

ALM – PROJECT BASED LEARNING

"Branch and Bound TSP for sales route planning"

DIKSHA YADAV	**2420030001**
KARISHMA SUTHAR	**2420030384**

COURSE INSTRUCTOR

Dr. J Sirisha Devi
Professor
Department of Computer Science and Engineering

TITLE

Traveling Salesman Problem using Branch and Bound for Optimal Sales Route Planning

The Traveling Salesman Problem (TSP) is a classical optimization problem where a salesman must visit all cities exactly once and return to the starting city while minimizing the total travel cost. This problem has direct applications in logistics, transportation, delivery routing, and manufacturing.

For example, consider a salesman who must visit five cities: A, B, C, D, and E. The distance between each pair of cities is known. A naive approach checks all possible routes (permutations), which becomes computationally expensive as the number of cities increases. Instead, using Branch and bound, we systematically explore and eliminate routes that cannot yield a better solution than the current best.

Key Objectives:

- Plan the most efficient sales route with minimal cost.
- Ensure all locations are visited exactly once.
- Return to the starting city.
- Use cost bounds to reduce unnecessary calculations and speed up finding the optimal path.

EXAMPLE

- Cities: A, B, C, D, E
- Distance Matrix:

	A	B	C	D	E
A	0	12	10	19	8
B	12	0	3	7	2
C	10	3	0	6	20
D	19	7	6	0	4
E	8	2	20	4	0

- Optimal Route (Example): A → B → C → D → E → A
- Minimum Cost: 33

This example demonstrates how the algorithm evaluates different paths but uses lower bound cost estimation to skip many unpromising routes, reaching the optimal tour faster.

ALGORITHM / PSEUDO CODE

Algorithm: Least Cost Branch and Bound for TSP

Step 1: Input distance matrix for n cities.
Step 2: Initialize the priority queue (min-heap).
Step 3: Start from the root node with the starting city.
Step 4: Calculate the lower bound using row and column reductions.
Step 5: Expand the node:

- If adding a city doesn't exceed the cost of the current best solution, branch further.
- Else, prune the node.
  Step 6: Keep track of the minimum cost and corresponding path.
  Step 7: Repeat until the queue is empty.
  Step 8: Output the optimal tour and its cost.

Pseudo Code

```
class Node {
   int level;          // current depth in the tree
   int cost;           // current path cost
   int bound;          // lower bound cost
   List<Integer> path;   // current path of cities
}

function calculateBound(int[][] dist, List<Integer> path, int n) {
   int bound = 0;

   // 1. Add current path cost
```

```
    for (int i = 0; i < path.size() - 1; i++) {
        bound += dist[path.get(i)][path.get(i + 1)];
    }

    // 2. Estimate minimal additional cost for remaining cities
    for (int i = 0; i < n; i++) {
        if (!path.contains(i)) {
            int minEdge = Integer.MAX_VALUE;
            for (int j = 0; j < n; j++) {
                if (i != j) {
                    minEdge = Math.min(minEdge, dist[i][j]);
                }
            }
            bound += minEdge;
        }
    }

    return bound;
}

function branchAndBoundTSP(int[][] dist, int n) {
    PriorityQueue<Node> pq = new PriorityQueue<>((a, b) -> a.bound - b.bound);
    int bestCost = Integer.MAX_VALUE;
    List<Integer> bestPath = new ArrayList<>();

    // Start from city 0
    Node root = new Node();
    root.path = new ArrayList<>();
    root.path.add(0);
```

```
root.cost = 0;
root.bound = calculateBound(dist, root.path, n);
root.level = 0;
pq.add(root);

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

   // Only expand if promising
   if (u.bound < bestCost) {
      for (int city = 0; city < n; city++) {
         if (!u.path.contains(city)) {
            Node v = new Node();
            v.path = new ArrayList<>(u.path);
            v.path.add(city);
            v.cost = u.cost + dist[u.path.get(u.path.size() - 1)][city];
            v.bound = calculateBound(dist, v.path, n);
            v.level = u.level + 1;

            // If all cities are visited, check total cost
            if (v.level == n - 1) {
               int totalCost = v.cost + dist[city][0];
               if (totalCost < bestCost) {
                  bestCost = totalCost;
                  v.path.add(0);  // return to start
                  bestPath = v.path;
               }
            } else if (v.bound < bestCost) {
               pq.add(v);
```

```
            }
          }
        }
      }
    }

    print("Minimum Cost: " + bestCost);
    print("Best Path: " + bestPath);
  }
```

TIME COMPLEXITY

The time complexity of the TSP using Branch and Bound depends on how effectively the algorithm prunes the search tree.

- Worst Case: O(n!) — when pruning is minimal.
- Average Case: Significantly better due to bounding.
- Bound Calculation: Each node calculation involves $O(n^2)$ operations due to cost matrix reduction.
- Total Nodes Explored: Much less than the brute-force approach because of pruning.

This makes the algorithm far more efficient than brute force for moderate values of n.

SPACE COMPLEXITY

- Priority Queue: Stores nodes of the search tree.
- Each node contains:
  - Current path → O(n)
  - Reduced matrix → $O(n^2)$
- Total Space: O (n^2 * number_of_nodes_in_PQ) in worst case.

Since many branches are pruned early, the actual memory used is often far less than the worst case.