Practical 9: Branch & Bound – Travelling Salesman Problem

Aim

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.

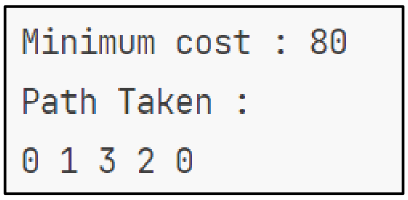
Algorithm

1. Initialize the algorithm by setting the initial lower bound to infinity and generating an initial feasible solution using a heuristic approach.
2. Create an empty priority queue to store the subproblems, with the highest priority given to the subproblems with the lowest lower bound.
3. Create the root node of the search tree by assigning the initial solution to it.
4. While the priority queue is not empty, do the following:
   a. Pop the subproblem with the lowest lower bound from the priority queue.
   b. If the subproblem represents a complete tour, update the lower bound and optimal solution if necessary.
   c. Otherwise, generate two child subproblems by selecting an unvisited city and creating two new tours, one that includes the city and one that does not. Calculate the lower bounds of the two child subproblems and add them to the priority queue.
5. If the optimal solution has been found, return it. Otherwise, return the best solution found.

Program

```
import java.util.*;
public class BranchNBound {

    // Stores number of cities
    static int N = 4;

    // Stores the final solution
    static int final_path[] = new int[N + 1];

    // Keeps track of the visited nodes
    static boolean visited[] = new boolean[N];

    // Stores minimum weight of shortest path.
    static int final_res = Integer.MAX_VALUE;

    // Function to copy temporary solution to final solution
    static void copyToFinal(int curr_path[]) {
        for (int i = 0; i < N; i++) {
            final_path[i] = curr_path[i];
        }
        final_path[N] = curr_path[0];
    }

    // Function to find the minimum edge cost having an end at the vertex i
    static int firstMin(int adj[][], int i) {
        int min = Integer.MAX_VALUE;
        for (int k = 0; k < N; k++) {
            if (adj[i][k] < min && i != k) {
                min = adj[i][k];
            }
        }
        return min;
    }

    // function to find the second minimum edge cost having an end at the vertex i
    static int secondMin(int adj[][], int i) {
        int first = Integer.MAX_VALUE, second = Integer.MAX_VALUE;
        for (int j = 0; j < N; j++) {
            if (i == j) {
                continue;
            }
            if (adj[i][j] <= first) {
                second = first;
                first = adj[i][j];
            } else if (adj[i][j] <= second && adj[i][j] != first) {
                second = adj[i][j];
            }
        }
        return second;
    }

    // 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[]

    static void TSPRec(int adj[][], int curr_bound, int curr_weight, int level, int curr_path[]) {
        // 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
                int curr_res = curr_weight +
                        adj[curr_path[level - 1]][curr_path[0]];

                // Update final result and final path if
                // current result is better.
                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 (int i = 0; i < N; i++) {
            // 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 && visited[i] == false) {
                int 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);
                }

                // 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
                Arrays.fill(visited, false);
                for (int j = 0; j <= level - 1; j++) {
                    visited[curr_path[j]] = true;
                }
            }
        }
    }

    // This function sets up final_path[]
    static void TSP(int adj[][]) {
        int curr_path[] = new int[N + 1];

        // 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
        int curr_bound = 0;
        Arrays.fill(curr_path, -1);
        Arrays.fill(visited, false);

        // Compute initial bound
        for (int i = 0; i < N; i++) {
            curr_bound += (firstMin(adj, i) + secondMin(adj, i));
        }

        // Rounding off the lower bound to an integer
        curr_bound = (curr_bound == 1) ? curr_bound / 2 + 1 : 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);
    }

    // Driver code
    public static void main(String[] args) {
        // Adjacency matrix for the given graph
        int adj[][] = {
                { 0, 10, 15, 20 },
                { 10, 0, 35, 25 },
                { 15, 35, 0, 30 },
                { 20, 25, 30, 0 } };

        TSP(adj);

        System.out.print("Minimum cost : " + final_res);
        System.out.println("\nPath Taken : ");
        for (int i = 0; i <= N; i++) {
            System.out.printf(final_path[i] + " ");
        }
    }
}
```

Output:



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

Analysis of Algorithm

Time Complexity:

The worst-case time complexity of the algorithm is **O((n-1)!),** where n is the number of cities in the input graph. This is because the algorithm involves exploring all possible permutations of the cities in order to find the optimal tour. However, in practice, the algorithm can be much faster due to the use of pruning techniques (e.g., calculating bounds and discarding subproblems that cannot lead to an optimal solution). The best-case time complexity occurs when the optimal solution is found in the first branch of the search tree, which takes **O(n)** time to compute. This can occur when the input graph has a special structure that allows the algorithm to quickly determine the optimal tour.

Space Complexity:

The space complexity of the algorithm depends on the number of subproblems that are generated and stored in memory during the search. In the worst case, the algorithm may need to store **O((n-1)!)** subproblems, which can be very large for large values of n.

However, in practice, the space complexity can be reduced by using techniques such as depth-first search or iterative deepening to limit the number of subproblems that are stored in memory at any given time.

Overall, the Branch and Bound algorithm for TSP is a powerful optimization technique that can find the optimal tour for small to medium-sized input graphs. However, for very large input graphs, the time and space complexity can become prohibitive, and alternative techniques such as heuristics or approximation algorithms may be more appropriate.