**Name**: Ravi Jamanbhai Makwana

**Roll No**: 21BCP418

Div-6, G-12

**Assignment-9 (Branch and Bound)**

**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.

Solve this problem using branch and bound technique.

**Theory:**

* Branch and bound is a widely used algorithmic technique for solving optimization problems, particularly in discrete optimization. The basic idea of branch and bound is to explore the search space systematically by dividing it into smaller subproblems, solving each subproblem separately, and then combining the solutions to obtain the overall solution.
* The branch and bound algorithm start by solving the entire problem, and then splits it into smaller subproblems, which are called branches. Each branch represents a subset of the original problem, and can be further divided into smaller subproblems. The algorithm assigns a lower bound to each branch based on the best possible solution found so far, and then determines whether the branch can be pruned (i.e., eliminated from further consideration) based on this lower bound.
* If a branch cannot be pruned, the algorithm continues to explore it by recursively splitting it into smaller subproblems, assigning lower bounds to each of the resulting branches, and checking for pruning conditions. The algorithm continues this process until a solution is found or all branches have been pruned.
* Branch and bound can be used for both maximization and minimization problems, and is particularly useful for problems where the search space is very large or where it is difficult to find an optimal solution through other means. However, the efficiency of branch and bound depends on the quality of the lower bounds used, as well as the branching strategy used to divide the search space.

**Algorithm:**

1. Initialize a priority queue Q to contain the root node of the search tree.

2. Initialize the current best solution to be a very large value (for minimization problems) or a very small value (for maximization problems).

3. While Q is not empty:

a. Dequeue the node with the highest priority from Q.

b. If the node is a leaf node, update the current best solution if the node's value is better than the current best solution.

c. Otherwise, generate all child nodes of the current node by applying a branching rule.

d. For each child node, compute a lower bound on the optimal solution for that node.

e. If the lower bound is worse than the current best solution, prune the node and do not enqueue it.

f. Otherwise, enqueue the node onto Q with priority equal to the lower bound.

4. Return the current best solution found.

**Program:**

import java.util.\*;

public class tsps {

    static int N;

    static int[][] adj;

    static int final\_path[] = new int[N + 1];

    static boolean visited[] = new boolean[N];

    static int min\_cost = Integer.MAX\_VALUE;

    static void copyToFinal(int path[]) {

        for (int i = 0; i < N; i++)

            final\_path[i] = path[i];

        final\_path[N] = path[0];

    }

    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;

    }

    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;

    }

    static void TSPRec(int adj[][], int bound, int weight, int level, int path[]) {

        if (level == N) {

            if (adj[path[level - 1]][path[0]] != 0) {

                int curr\_res = weight + adj[path[level - 1]][path[0]];

                if (curr\_res < min\_cost) {

                    copyToFinal(path);

                    min\_cost = curr\_res;

                }

            }

            return;

        }

        for (int i = 0; i < N; i++) {

            if (adj[path[level - 1]][i] != 0 && visited[i] == false) {

                int temp = bound;

                weight += adj[path[level - 1]][i];

                if (level == 1) {

                    bound -= ((firstMin(adj, path[level - 1]) + firstMin(adj, i)) / 2);

                } else {

                    bound -= ((secondMin(adj, path[level - 1]) + firstMin(adj, i)) / 2);

                }

                if (bound + weight < min\_cost) {

                    path[level] = i;

                    visited[i] = true;

                    TSPRec(adj, bound, weight, level + 1, path);

                }

                weight -= adj[path[level - 1]][i];

                bound = temp;

                Arrays.fill(visited, false);

                for (int j = 0; j <= level - 1; j++) {

                    visited[path[j]] = true;

                }

            }

        }

    }

    static void TSP(int adj[][]) {

        int path[] = new int[N + 1];

        int bound = 0;

        Arrays.fill(path, -1);

        Arrays.fill(visited, false);

        for (int i = 0; i < N; i++)

            bound += (firstMin(adj, i) + secondMin(adj, i));

        bound = (bound == 1) ? bound / 2 + 1 : bound / 2;

        visited[0] = true;

        path[0] = 0;

        TSPRec(adj, bound, 0, 1, path);

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the number of cities: ");

        N = sc.nextInt();

        adj = new int[N][N];

        final\_path = new int[N + 1];

        visited = new boolean[N];

        min\_cost = Integer.MAX\_VALUE;

        System.out.println("Enter the adjacency matrix:");

        for (int i = 0; i < N; i++) {

            for (int j = 0; j < N; j++) {

                adj[i][j] = sc.nextInt();

            }

        }

        TSP(adj);

        System.out.println("\nMinimum cost: " + min\_cost);

        System.out.println("Final Path:");

        for (int i = 0; i <= N; i++) {

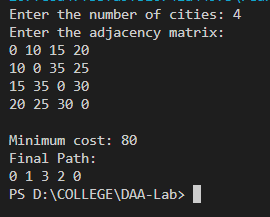
            System.out.print(final\_path[i] + " ");

        }

    }

}

**Output:**



**Analysis:**

**Time Complexity Analysis:**

**Best Case:**

The best-case time complexity of branch and bound is typically O(1), which occurs when the optimal solution is found in the first node of the search tree.

**Average Case:**

The average-case time complexity of branch and bound is difficult to analyse, as it depends on the specific problem instance and the quality of the lower bounds. In general, the time complexity can be bounded by O(bd), where b is the branching factor and d is the depth of the search tree.

**Worst Case:**

The worst-case time complexity of branch and bound is typically exponential in n, and can be expressed as O(bn). This occurs when the branching factor is high, and the optimal solution is located in a leaf node deep in the search tree. However, in practice, the worst-case complexity is rarely encountered, as the algorithm can often prune large portions of the search space using the lower bounding method.