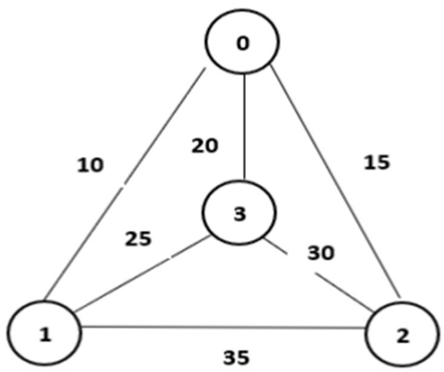
EXPERIMENT 9

20CP209P – Design and Analysis of Algorithm Lab

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.

For example, consider the following graph:



A Travelling Salesman Problem (TSP) tour in the graph is 0 − 1 − 3 − 2 − 0. The cost of the tour is 10 + 25 + 30 + 15 = 80

Code:

Travelling Salesman Problem (Branch and Bound):

```
#include <iostream>
#include <vector>
#include <limits> // Use <limits> for numeric_limits
#include <algorithm>
#include <numeric>
#include <vector>
#include <queue> // For priority queue

#define fr(i, a, b) for (int i = a; i < b; i++)
#define pii pair<int,int>

using namespace std;

// Use numeric_limits for infinity for better type safety and clarity
const int INF = numeric_limits<int>::max();

// Structure to represent a node in the state-space tree
struct TSPNode {
```

```
    vector<int> path;      // Path taken so far
    vector<bool> visited;   // Visited cities marker
    int cost;              // Cost of the path so far
    int lower_bound;       // Estimated lower bound for the total tour cost
    int level;             // Number of cities visited (path.size())

    // Constructor
    TSPNode(int n) : visited(n, false), cost(0), lower_bound(0), level(0) {}

    // Custom comparator for the priority queue (min-heap based on lower_bound)
    bool operator>(const TSPNode& other) const {
        return lower_bound > other.lower_bound;
    }
};

// Function to calculate the lower bound for a given node
// A simple lower bound: current cost + sum of minimum outgoing edge from each unvisited city
int calculate_lower_bound(const TSPNode& node, int n, const vector<vector<int>>& graph)
{
    int bound = node.cost;
    int last_city = node.path.back();

    for (int i = 0; i < n; ++i)
    {
        // If city 'i' is unvisited OR it's the starting city and we need to return
        if (!node.visited[i])
        {
            // Find the minimum cost edge leaving city 'i' to any *other* city
            int min_edge = INF;
            // If it's the last node in the path, find min edge to unvisited or start
            if (i == last_city)
            {
                for (int j = 0; j < n; ++j)
                {
                    // Must go to an unvisited city OR back to start if it's the last hop
                    if (j == node.path[0] && node.level == n)
                    { // Check if it's the last step
                        if (graph[i][j] != INF)
                        {
                          min_edge = min(min_edge, graph[i][j]);
                        }
                    }
                    else if (!node.visited[j] && i != j && graph[i][j] != INF)
                    {
                      min_edge = min(min_edge, graph[i][j]);
                    }
                }
            }
            else
            { // For other unvisited cities, find the absolute minimum outgoing edge
```

```
            for(int j=0; j<n; ++j)
            {
                if (i != j && graph[i][j] != INF)
                {
                    min_edge = min(min_edge, graph[i][j]);
                }
            }
         }

        // If an unvisited city has no way out (shouldn't happen in complete graph)
        // or if it's the last node with no path back to start
        if (min_edge == INF)
        {
            return INF; // This path is infeasible or bound calculation failed
        }
        bound += min_edge;
    }
}
// A slightly simpler version (often used, might be less tight but still valid):
// Sum minimum edge cost for all unvisited nodes, regardless of where they go
/*
int simpler_bound = node.cost;
for(int i=0; i<n; ++i) {
    if(!node.visited[i]) {
        int min_edge = INF;
        for(int j=0; j<n; ++j) {
            if(i != j && graph[i][j] != INF) {
                min_edge = min(min_edge, graph[i][j]);
            }
        }
        if(min_edge == INF) return INF; // Infeasible
        simpler_bound += min_edge;
    }
}
// Add cost from last visited node back to start if it's the last leg needed
if (node.level == n -1 && !node.visited[0]) { // Check if only start is left
    int last = node.path.back();
    if(graph[last][0] != INF) {
         simpler_bound += graph[last][0]; // Already partially counted, tricky
    } else {
         return INF; // No path back to start
    }
}
return simpler_bound; // Let's use the simpler bound for clarity
*/

// Let's stick to the slightly more accurate one calculated first.
// If the path is nearly complete, the last leg must go back to start
if (node.level == n)
{
```

```
        int return_cost = graph[last_city][node.path[0]];
        if (return_cost == INF) return INF; // No path back
        // The bound IS the final cost here
        return node.cost + return_cost;
    }

    return bound;
}

// --- Simplified Lower Bound Implementation ---
// Calculate the lower bound: cost so far + sum of min edges from unvisited nodes
int calculate_simplified_lower_bound(int current_cost, int current_city, int n, const vector<bool>&
visited, const vector<vector<int>>& graph)
{
    int bound = current_cost;

    for (int i = 0; i < n; ++i)
    {
        // Consider unvisited cities
        if (!visited[i])
        {
            int min_val = INF;
            for (int j = 0; j < n; ++j)
            {
                // Find the cheapest edge leaving this unvisited city 'i'
                // The destination 'j' can be any other city (visited or unvisited)
                // or the starting city if 'i' is the last city visited in a partial tour
                if (i != j && graph[i][j] != INF)
                {
                   // Simple version: min edge to *any* other node
                    min_val = min(min_val, graph[i][j]);
                }
            }
             if(min_val == INF) return INF; // This city is stranded
             bound += min_val;

        }
    }

    // Special handling for the edge FROM the current_city
    // Find the minimum edge from current_city to an UNVISITED city
    int min_from_current = INF;
    bool can_reach_unvisited = false;
    for(int j = 0; j < n; ++j) {
        if (!visited[j] && graph[current_city][j] != INF)
        {
            min_from_current = min(min_from_current, graph[current_city][j]);
            can_reach_unvisited = true;
        }
    }
```

```
    // If we can't reach any unvisited node from current city (and tour not complete)
    // it might mean we only need to go back to start
    if (!can_reach_unvisited && count(visited.begin(), visited.end(), false) == 1 && !visited[0])
    {
      if (graph[current_city][0] != INF)
      {
        // The only unvisited node is the start node
        // The lower bound should include the cost back to start
        // Note: The loop above already added min cost *from* start (if unvisited)
        // Let's refine the loop logic slightly:
        // bound = current_cost
        // sum min outgoing edge for all nodes k (!visited)
        // sum min incoming edge for all nodes k (!visited)
        // return bound = current_cost + (sum_min_out + sum_min_in) / 2 (Held-Karp idea - complex)

        // Stick to simpler: Sum of min outgoing from unvisited
        // Let's recalculate simpler bound:
        bound = current_cost;
        for(int i=0; i<n; ++i)
        {
          // For the current node, find min edge to UNVISITED node
          if(i == current_city)
          {
            int min_edge_curr = INF;
             for(int j=0; j<n; ++j)
             {
               if(!visited[j] && graph[i][j] != INF)
               {
                 min_edge_curr = min(min_edge_curr, graph[i][j]);
               }
             }
             // If only start node is left unvisited
             if (min_edge_curr == INF && count(visited.begin(), visited.end(), false) == 1 &&
!visited[0])
             {
                 if(graph[i][0] != INF)
                 {
                   min_edge_curr = graph[i][0];
                 }
                 else
                 {
                   return INF; // Cannot return to start
                 }
             }
             else if (min_edge_curr == INF && count(visited.begin(), visited.end(), false) > 0)
             {
                 return INF; // Cannot reach any other unvisited node
             }
```

```
                if(min_edge_curr != INF) bound += min_edge_curr; // Add cost from current node
onwards

            }
            // For OTHER unvisited nodes, find the absolute minimum outgoing edge
            else if (!visited[i])
            {
                int min_edge_other = INF;
                for(int j=0; j<n; ++j)
                {
                    if(i != j && graph[i][j] != INF)
                    {
                        min_edge_other = min(min_edge_other, graph[i][j]);
                    }
                }
                if(min_edge_other == INF) return INF; // Stranded node
                bound += min_edge_other;
            }
        }
        return bound;

      }
      else
      {
        return INF; // Can't get back to start
      }
    }

    return bound; // Return the calculated simple bound
}

int main(void)
{
    // Example Graph (Cost Matrix)
    // graph[i][j] = cost from city i to city j
    // Use INF if no direct path (or for diagonal i == j)
     vector<vector<int>> graph = {
       {INF, 10, 8, 9, 7},
       {10, INF, 10, 5, 6},
       {8, 10, INF, 8, 9},
       {9, 5, 8, INF, 6},
       {7, 6, 9, 6, INF}
    };
    int n = graph.size(); // Number of cities

    // Use graph2 for testing
    // vector<vector<int>> graph2 = {
    //     {INF, 10, 15, 20},
    //     {5, INF, 9, 10},
    //     {6, 13, INF, 12}, // Changed 9 to 13 based on common examples for cost 35
```

```
//    {8, 8, 9, INF}
// };
// vector<vector<int>> graph = graph2; // Uncomment to use graph2
// n = graph.size();

// Priority Queue for B&B nodes (min-heap based on lower_bound)
priority_queue<TSPNode, vector<TSPNode>, greater<TSPNode>> pq;

// Global minimum cost found so far
int min_cost = INF;
vector<int> final_path;

// Create the root node (starting at city 0)
TSPNode root(n);
root.level = 1;
root.path.push_back(0); // Start at city 0
root.visited[0] = true;
root.cost = 0;
 // Calculate initial lower bound (sum of min edges from all nodes) - Adjusted calculation
root.lower_bound = calculate_simplified_lower_bound(root.cost, 0, n, root.visited, graph);
// root.lower_bound = 0; // Initial cost is 0
// for(int i=0; i<n; ++i) {
//    int min_row = INF;
//    for (int j=0; j<n; ++j) {
//       if (i != j && graph[i][j] != INF) {
//          min_row = min(min_row, graph[i][j]);
//       }
//    }
//     if(min_row == INF && n > 1) { // Check for infeasible graphs early
//        cout << "Graph is not strongly connected or has isolated nodes." << endl;
//        return 1;
//     }
//     if (min_row != INF) root.lower_bound += min_row;
// }

if (root.lower_bound != INF) {
   pq.push(root);
} else if (n > 0) {
    cout << "Initial lower bound is INF. Problem might be infeasible." << endl;
}

cout << "Starting Branch and Bound TSP..." << endl;
if (n <= 1) {
    cout << "Minimum Cost: 0" << endl;
    cout << "Path: 0" << endl;
    return 0;
}

// Branch and Bound main loop
while (!pq.empty()) {
```

```
// Get the node with the lowest lower_bound
TSPNode current_node = pq.top();
pq.pop();

// --- Pruning Step ---
// If the current node's lower bound is already worse than the best solution found, prune it.
if (current_node.lower_bound >= min_cost) {
  // cout << "Pruning node. Lower bound (" << current_node.lower_bound
  //      << ") >= min_cost (" << min_cost << ")" << endl;
  continue;
}

// --- Goal Check ---
// If all cities have been visited (path length = n)
if (current_node.level == n) {
  // We need to add the cost of returning to the starting city (city 0)
  int last_city = current_node.path.back();
  int return_cost = graph[last_city][0]; // Cost from last city back to start

  if (return_cost != INF) {
    int total_cost = current_node.cost + return_cost;

    // Found a new best solution
    if (total_cost < min_cost) {
      min_cost = total_cost;
      final_path = current_node.path;
      // Add the starting city to the end to show the cycle
      final_path.push_back(0);
      cout << "Found new best solution: Cost = " << min_cost << ", Path = ";
       for(int city : final_path) cout << city << " "; cout << endl;
    }
  }
  // No further branching needed from a complete tour node
  continue;
}

// --- Branching Step ---
// Explore neighbors (unvisited cities)
int current_city = current_node.path.back();
for (int next_city = 0; next_city < n; ++next_city) {
  // If the next city is not visited and there's an edge
  if (!current_node.visited[next_city] && graph[current_city][next_city] != INF)
  {
    // Create a child node representing the extended path
    TSPNode child_node = current_node; // Copy parent state

    child_node.level++;
    child_node.path.push_back(next_city);
    child_node.visited[next_city] = true;
    child_node.cost = current_node.cost + graph[current_city][next_city];
```

```
            // Calculate the lower bound for the child node
            child_node.lower_bound = calculate_simplified_lower_bound(child_node.cost, next_city, n,
child_node.visited, graph);

            // cout << "  Exploring edge " << current_city << "->" << next_city
            //      << ", Cost: " << child_node.cost
            //      << ", Lower Bound: " << child_node.lower_bound << endl;

            // --- Pruning before adding to queue ---
            // Only add the child to the queue if its bound is promising
            if (child_node.lower_bound < min_cost) {
                pq.push(child_node);
            } else {
                // cout << "  Pruning child path (bound " << child_node.lower_bound << " >= min_cost "
<< min_cost << ")" << endl;
            }
          }
        }
    }

    // Output the result
    if (min_cost == INF) {
        cout << "\nNo feasible solution found." << endl;
    } else {
        cout << "\n-----------------------------------" << endl;
        cout << "Optimal Minimum Cost: " << min_cost << endl;
        cout << "Optimal Path: ";
        for (int i = 0; i < final_path.size(); ++i) {
            cout << final_path[i] << (i == final_path.size() - 1 ? "" : " -> ");
        }
        cout << endl;
        cout << "-----------------------------------" << endl;
    }

    return 0;
}
```

Output:

```
PS B:\sem4\23bcp153_daa\lab9> g++ tspbnbgem.cpp -o tspbnbgem
PS B:\sem4\23bcp153_daa\lab9> ./tspbnbgem
Starting Branch and Bound TSP...
Found new best solution: Cost = 34, Path = 0 4 1 3 2 0

----------------------------------
Optimal Minimum Cost: 34
Optimal Path: 0 -> 4 -> 1 -> 3 -> 2 -> 0
----------------------------------
PS B:\sem4\23bcp153_daa\lab9>
```