Assignment No. 4

Aim: Implement Travelling Salesman problem using branch and bound technique.

Theory:

Travelling Salesman Problem is based on a real life scenario, where a salesman from a company has to start from his own city and visit all the assigned cities exactly once and return to his home till the end of the day. The exact problem statement goes like this,

"Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point." There are two important things to be cleared about in this problem statement,

- Visit every city exactly once
- Cover the shortest path

❖ **Branch and Bound**

Before going into the details of the branch and bound method lets focus on some important terms for the same,

- State Space Search Method - Remember the word sample space from probability theory ? That was a set of all the possible sample outputs. In a similar way,state space means a set of states that a problem can be in. The set of states forms a graph where two states are connected if there is an operation that can be performed to transform the first state into the second.
  In a lighter note, this is just a set of some objects, which has some properties/characteristics, like in this problem, we have a node, it has a cost associated to it. The entire state space can be represented as a tree known as state space tree, which has the root and the leaves as per the normal tree, which are in terms of the elements of the state space having the given graph node and a cost associated to it.
- E-node - Expanded node or E node is the node which has been expanded. As we know a tree can be expanded using both BFS(Breadth First Search) and DFS(Depth First Search),all the expanded nodes are known as E-nodes.
- Live-node - A node which has been generated and all of whose children have not yet been expanded is called a live-node.
- Dead-node - If a node can't be expanded further, it's known as a dead-node.

The word, Branch and Bound refers to all the state space search methods in which we generate the children of all the expanded nodes, before making any live node as an expanded one. In this method, we find the most promising node and expand it. The term promising node means, choosing a node that can expand and give us an optimal solution. We start from the root and expand the tree until unless we approach an optimal (minimum cost in case of this problem) solution.

Code:

```
#include <bits/stdc++.h>
using namespace std;

// number of total nodes
#define N 5
#define INF INT_MAX

class Node
{
public:
    vector<pair<int, int>> path;
    int matrix_reduced[N][N];
    int cost;
    int vertex;
    int level;
};

Node* newNode(int matrix_parent[N][N], vector<pair<int, int>> const &path,int
level, int i, int j)
{
    Node* node = new Node;
    node->path = path;
    if (level != 0)
        node->path.push_back(make_pair(i, j));
    memcpy(node->matrix_reduced, matrix_parent,
        sizeof node->matrix_reduced);
    for (int k = 0; level != 0 && k < N; k++)
    {
        node->matrix_reduced[i][k] = INF;
        node->matrix_reduced[k][j] = INF;
```

```
    }

    node->matrix_reduced[j][0] = INF;
    node->level = level;
    node->vertex = j;
    return node;
}

void reduce_row(int matrix_reduced[N][N], int row[N])
{
    fill_n(row, N, INF);

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            if (matrix_reduced[i][j] < row[i])
                row[i] = matrix_reduced[i][j];

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            if (matrix_reduced[i][j] != INF && row[i] != INF)
                matrix_reduced[i][j] -= row[i];
}

void reduce_column(int matrix_reduced[N][N], int col[N])
{
    fill_n(col, N, INF);

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            if (matrix_reduced[i][j] < col[j])
                col[j] = matrix_reduced[i][j];

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            if (matrix_reduced[i][j] != INF && col[j] != INF)
                matrix_reduced[i][j] -= col[j];
}

int cost_calculation(int matrix_reduced[N][N])
{
```

```
    int cost = 0;

    int row[N];
    reduce_row(matrix_reduced, row);

    int col[N];
    reduce_column(matrix_reduced, col);

    for (int i = 0; i < N; i++)
        cost += (row[i] != INT_MAX) ? row[i] : 0,
            cost += (col[i] != INT_MAX) ? col[i] : 0;

    return cost;
}

void printPath(vector<pair<int, int>> const &list)
{
    for (int i = 0; i < list.size(); i++)
        cout << list[i].first + 1 << " -> "
             << list[i].second + 1 << endl;
}

class comp {
public:
    bool operator()(const Node* lhs, const Node* rhs) const
    {
        return lhs->cost > rhs->cost;
    }
};

int solve(int adjacensyMatrix[N][N])
{
    priority_queue<Node*, std::vector<Node*>, comp> pq;
    vector<pair<int, int>> v;
    Node* root = newNode(adjacensyMatrix, v, 0, -1, 0);
    root->cost = cost_calculation(root->matrix_reduced);
    pq.push(root);
    while (!pq.empty())
    {
        Node* min = pq.top();
```

```
        pq.pop();
        int i = min->vertex;
        if (min->level == N - 1)
        {
            min->path.push_back(make_pair(i, 0));
            printPath(min->path);
            return min->cost;
        }

        for (int j = 0; j < N; j++)
        {
            if (min->matrix_reduced[i][j] != INF)
            {
                Node* child = newNode(min->matrix_reduced, min->path,
                    min->level + 1, i, j);

                child->cost = min->cost + min->matrix_reduced[i][j]
                            + cost_calculation(child->matrix_reduced);

                pq.push(child);
            }
        }

        delete min;
    }
return 0;
}

int main()
{
    int adjacensyMatrix[N][N] =
    {
        { INF, 20, 30, 10, 11 },
        { 15,  INF, 16,  4,  2 },
        { 3,   5,  INF, 2,  4 },
        { 19,   6,   18,   INF, 3 },
        { 16,   4,   7,   16, INF }
    };

    cout << "\nCost is " << solve(adjacensyMatrix);
```

```
    return 0;
}
```

Output:

```
Cost is 1 -> 4
4 -> 2
2 -> 5
5 -> 3
3 -> 1
28

...Program finished with exit code 0
Press ENTER to exit console.
```