Name : Pratik Kanhaiya Rathod
Roll : 321056
PRN : 22010885
Batch : A2

Assignment No. 4

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

Theory:

What is branch and bound technique?

1) Branch and bound is an algorithm used to solve combinatorial optimization problems. These problems are usually augmented in terms of time complexity and issues that require exploring all permutations and combinations possible in a worst-case scenario.
2) The branch and bound calculator is typically used to solve problems that are not solvable in polynomial time, such as network flow issues, crew scheduling, and production planning. However, it can be better understood as a backtracking tool using the state space tree.
3) As the name suggests, BnB explores branches or nodes under a particular subset of solutions. The parent node is considered to have all the possible solutions to the said problem. However, before the candidate solutions to the issue are computed or enumerated, the upper and lower limit binds the optimal solution.
4) The branch and bound method were put forth first by Alisa Land and Alison Doig from the London School of Economics in 1960 as a discrete programming solution. However, the name "BnB" was first used officially in a study on traveling salesman issues.

Travelling salesman problem:

1) The traveling salesman problem (TSP) is an algorithmic problem tasked with finding the shortest route between a set of points and locations that must be visited. In the problem statement, the points are the cities a salesperson might visit. The salesman's goal is to keep both the travel costs and the distance traveled as low as possible.

2) Focused on optimization, TSP is often used in computer science to find the most efficient route for data to travel between various nodes. Applications include identifying network or hardware optimization methods. It was first described by Irish mathematician W.R. Hamilton and British mathematician Thomas Kirkman in the 1800s through the creation of a game that was solvable by finding a Hamilton cycle, which is a non-overlapping path between all nodes.

Implementation of Travelling Salesman problem:

```
#include<bits/stdc++.h>
#define INF 99
#define cities 5
using namespace std;

int travel_path[cities] = {0}, pos = 0;
int been_there[cities] = {0};
int cost_matrix[cities][cities] ={
    {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}
};

int tsp(int matrix[cities][cities], int total_cost, int where_am_i){
    int least_cost_bound = INF;
    int travel_cost = 0;
    int new_matrix[cities][cities];
    int col[cities], row[cities];
    int keep_node;
    int keep_matrix[cities][cities];
    been_there[where_am_i] = 1;
    int recurtion_times = 0;

    cout<<"\n------------------\n";
    for(int i =0; i<cities; i++){
        for(int k = 0; k<cities; k++){
            if(matrix[i][k] == INF){
                cout<<"oo\t";
            }
            else{
                cout<<matrix[i][k]<<"\t";
            }
        }
        cout<<endl;
    }
    cout<<"\n------------------\n";
    travel_path[pos] = where_am_i+1;
    pos++;

    for(int i=0; i<cities; i++){
        if(been_there[i] != 1){
            recurtion_times++;
            //reconstruct array
            for(int k=0; k<cities; k++){
```

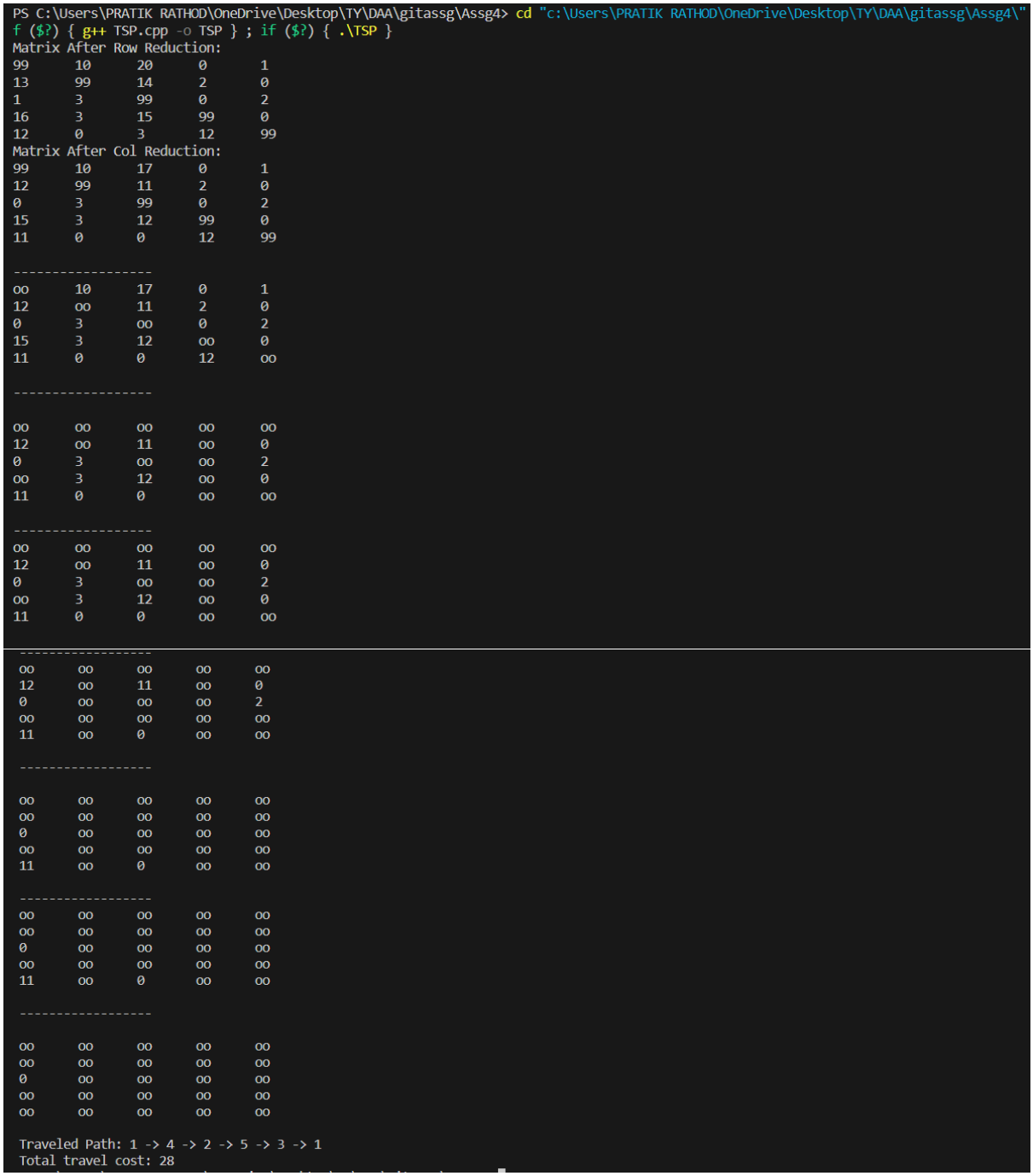
```
                for(int l=0; l<cities; l++){
                    if(k == where_am_i || l == i || (l == where_am_i && k ==
i) )
                        new_matrix[k][l] = INF;
                    else
                        new_matrix[k][l] = matrix[k][l];
                }
            }
    for(int k=0; k<cities; k++){
                row[k] = new_matrix[k][0];
                for(int l=0; l<cities; l++){
                    if(new_matrix[k][l] != INF){
                        if(new_matrix[k][l] < row[k])
                            row[k] = new_matrix[k][l];
                    }

                }
                if(row[k] == INF)
                    row[k] = 0;
                travel_cost += row[k];
            }
    if(travel_cost > 0){
                for(int k=0; k<cities; k++){
                    for(int l=0; l<cities; l++){
                        if(new_matrix[k][l] != INF)
                            new_matrix[k][l] = new_matrix[k][l] - row[k];
                        else
                            new_matrix[k][l] = new_matrix[k][l];
                    }
                }
    }
    for(int k=0; k<cities; k++){
                col[k] = new_matrix[0][k];
                for(int l=0; l<cities; l++){
                    if(new_matrix[l][k] != INF){
                        if(new_matrix[l][k] < col[k])
                            col[k] = new_matrix[l][k];
                    }
                }
                if(col[k] == INF)
                    col[k] = 0;
                travel_cost += col[k];
    }
    //making the reduced array subtracting the lowest value of every column
            if(travel_cost > 0){
                for(int k=0; k<cities; k++){
                    for(int l=0; l<cities; l++){
                        if(new_matrix[k][l] != INF){
```

```
                            new_matrix[k][l] = new_matrix[k][l] - col[l];
                        }else
                            new_matrix[k][l] = new_matrix[k][l];
                    }
                }
            }
            ///////////////

            if(least_cost_bound > total_cost + matrix[where_am_i][i] +
travel_cost){
                least_cost_bound = total_cost + matrix[where_am_i][i] +
travel_cost;
                //keep position in order to know which node has the lowest LCB
                keep_node = i;
                for(int k=0; k<cities; k++){
                    for(int l=0; l<cities; l++){
                        keep_matrix[k][l] = new_matrix[k][l];
                    }
                }
            }
            // printf("LCB: %d at node: %d\n", least_cost_bound, keep_node);
        }
        travel_cost = 0;
    }
    cout<<endl;
    for(int k=0; k<cities; k++){
        for(int f=0; f<cities; f++){
            if(keep_matrix[k][f] == INF)
                cout<<"oo\t";
            else
                cout<<keep_matrix[k][f]<<"\t";
        }
        cout<<endl;
    }
    if(recurtion_times < 2){
        travel_path[pos] = keep_node+1;
        return least_cost_bound;
    }else{
        // printf("total_cost recurive: %d\n", total_cost);
        total_cost = tsp(keep_matrix, least_cost_bound, keep_node);
    }

}

int main(){
    int reduced_matrix[cities][cities];
    int col[cities], row[cities];
    int total_cost = 0;
```

```
    //finding min of every row
    for(int i = 0; i<cities; i++){
        row[i] = cost_matrix[i][0];
        for (int j = 0; j < cities; j++)
        {
            if(cost_matrix[i][j] < row[i]){
                row[i] = cost_matrix[i][j];
            }
        }
        total_cost += row[i];
    }
    cout<<"Matrix After Row Reduction: "<<endl;
    for(int i = 0; i<cities; i++){
        for(int k = 0; k <cities; k++){
            if(cost_matrix[i][k] != INF) reduced_matrix[i][k] =
cost_matrix[i][k] - row[i];
            else reduced_matrix[i][k] = cost_matrix[i][k];
            cout<<reduced_matrix[i][k]<<"\t";
        }
        cout<<endl;
    }
    //finding min of every col
    for(int i = 0; i<cities; i++){
        col[i] = reduced_matrix[0][i];
        for(int j = 0; j<cities; j++){
            if(reduced_matrix[j][i] != INF){
                if(reduced_matrix[j][i] < col[i]){
                    col[i] = reduced_matrix[j][i];
                }
            }
        }
        total_cost += col[i];
    }
    cout<<"Matrix After Col Reduction: "<<endl;
    for(int i = 0; i<cities; i++){
        for(int k = 0; k<cities; k++){
            if(reduced_matrix[i][k] != INF){
                reduced_matrix[i][k] = reduced_matrix[i][k] - col[k];
            }
            else{
                reduced_matrix[i][k] = reduced_matrix[i][k];
            }
            cout<<reduced_matrix[i][k]<<"\t";
        }
        cout<<endl;
    }
    total_cost = tsp(reduced_matrix, total_cost, 0);
```

```
    cout<<"\nTraveled Path: ";
    for(int i = 0;i<cities;i++){
        cout<<travel_path[i]<<" -> ";
    }
    cout<<"1";
    cout<<"\nTotal travel cost: "<<total_cost;
    return 0;
}
                    //          %
                    //      /       \
                    //    /  \     /  \
```

Output :



Analysis:

- n is the number of cities.
- Time Complexity of TSP: O(n^2)
- Space Complexity: O(n^2) storing edges and O(2^n) stack space