Practical 8
Implementing Traveling Salesman Problem

Gahan M. Saraiya, 18MCEC10

18mcec10@nirmauni.ac.in

I. Introduction

Aim of this practical is to Implement **Traveling Salesman Problem (TSP** using Branch and Bound method.

Traveling Salesman Problem is defined as "*Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?*" It is an NP-hard problem.

- Branch and bound (**BB** or **B&B**) is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization.

II. Implementation

- Model Traveling Salesman Problem as an undirected weighted graph.
- consider cities as graph's vertices
- consider paths as graph's edges
- consider path's distance as edge's length (weight of edge)
- TSP is minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once.

I. Main Program - ***traveling_salesman_problem.c***

```c
//
↪ -----------------------------------------------------------------------------------------
// Author: Gahan Saraiya
// GiT: http://github.com/gahan9/
// StackOverflow: https://stackoverflow.com/users/story/7664524
// Website: http://gahan9.github.io/
//
↪ -----------------------------------------------------------------------------------------
// Implementing Traveling Salesman Problem using Branch and Bound

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```c
#include "../utils/constant.h"
#include "../utils/utility.h"

#define inf 2147483647

int cost = 0;

//int matrix[100][100], visited[100], number_of_cities, cost = 0;
int display_cost_matrix(int **array, int no_of_elements) {
    // display given array of given size(no. of elements require because sizeof()
     ↪ returns max bound value)
    write_log(": ");
    for (int i = 0; i < no_of_elements; i++) {
        printf("City-%d >>: ", i);
        for (int j = 0; j < no_of_elements; j++) {
            printf("%d ", array[i][j]);
        }
        printf("\n");
    }
    return 0;
}

int get_nearest_city(int c, int **cost_matrix, int *visited_status, int
 ↪ number_of_cities) {
    int nearest_neighbour_city = inf;
    int min = inf, temp;
    for (int i = 0; i < number_of_cities; i++) {
        if ((cost_matrix[c][i] != 0) && (visited_status[i] == 0)) {
            if (cost_matrix[c][i] < min) {
                min = cost_matrix[i][0] + cost_matrix[c][i];
            }
            temp = cost_matrix[c][i];
            nearest_neighbour_city = i;
        }
    }
    if (min != inf)
        cost += temp;
    return nearest_neighbour_city;
}

int min_cost_calc(int city, int **cities, int *visited_cities, int
 ↪ number_of_cities) {
    int i, nearest_neighbour_city;
    visited_cities[city] = 1;
    printf("%d ===> ", city + 1);
    nearest_neighbour_city = get_nearest_city(city, cities, visited_cities,
     ↪ number_of_cities);

    if (nearest_neighbour_city == inf) {
        nearest_neighbour_city = 0;
        printf("%d", nearest_neighbour_city + 1);
        cost += cities[city][nearest_neighbour_city];
        return 1;
    }
    return min_cost_calc(nearest_neighbour_city, cities, visited_cities,
     ↪  number_of_cities);
}

int main() {
    int number_of_cities;
    printf("\nEnter Number of Cities: \n");
    scanf("%d", &number_of_cities);
    int **cities = malloc(number_of_cities * sizeof(int *));
    int *visited_cities = malloc(number_of_cities * sizeof(int *));
    for (int i = 0; i < number_of_cities; i++) {
        cities[i] = malloc(number_of_cities * sizeof(int));
    }


    printf("\nEnter Cost Matrix for travelling through %d cities: \n",
     ↪  number_of_cities);
    for (int i = 0; i < number_of_cities; i++) {
//        printf("\n Enter cost from city 1# : %d\n", i + 1);
        for (int j = 0; j < number_of_cities; j++)
            if (i == j) {
//                cities[i][j] = inf;
                cities[i][j] = 0;
            } else {
                cities[i][j] = 1 + rand() % 9;
            }
//            scanf("%d", &cities[i][j]);
        visited_cities[i] = 0;
    }

    printf("\nThe Cost Matrix is:\n");
    display_cost_matrix(cities, number_of_cities);

    printf("\nThe Path is:\n");
    min_cost_calc(0, cities, visited_cities, number_of_cities);
    printf("\nMinimum Cost of tour: -> %d", cost);
    return 1;
}
```

I.1 Output

```
Enter Number of Cities: 6

Enter Cost Matrix for travelling through 6 cities:

The Cost Matrix is:
City-0 >>: 0 2 8 15 1 10
City-1 >>: 5 0 19 19 3 5
City-2 >>: 6 6 0 2 8 2
City-3 >>: 12 16 3 0 8 17
City-4 >>: 12 5 3 14 0 13
City-5 >>: 3 2 17 19 16 0

The Path is:
1 ===> 6 ===> 5 ===> 4 ===> 3 ===> 2 ===> 1
Minimum Cost of tour: -> 54
```

I.2 Output

```
Enter Number of Cities:

Enter Cost Matrix for travelling through 7 cities:

The Cost Matrix is:
City-0 >>: 0 6 9 8 5 9 2
City-1 >>: 4 0 1 8 3 9 3
City-2 >>: 8 7 0 8 6 8 9
City-3 >>: 4 1 1 0 7 6 1
City-4 >>: 5 8 7 6 0 9 6
City-5 >>: 3 1 3 1 7 0 5
City-6 >>: 9 2 8 4 3 7 0

The Path is:
1 ===> 7 ===> 6 ===> 5 ===> 4 ===> 3 ===> 2 ===> 1
Minimum Cost of tour: -> 34
```