DAA Assignment-05

Nischay Nagar
IIT2019198

Abhishek Bithu
IIT2019199

Raj Chandra
IIT2019200

***Abstract*—In this report we designed a Dynamic Programming algorithm for solving the Travelling Salesmen Problem(TSP). .(For 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)**

I. Introduction

The travelling salesman problem (also called the traveling salesperson problem or TSP) asks the following question: ”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 in combinatorial optimization, important in theoretical computer science and operations research.

The travelling purchaser problem and the vehicle routing problem are both generalizations of TSP.

In the theory of computational complexity, the decision version of the TSP (where given a length L, the task is to decide whether the graph has a tour of at most L) belongs to the class of NP-complete problems. Thus, it is possible that the worst-case running time for any algorithm for the TSP increases superpolynomially (but no more than exponentially) with the number of cities.

II. Algorithm Design

Following is Naive algorithm:

- Consider city 1 as the starting and ending point.
- Generate all (n-1)! Permutations of cities.
- Calculate cost of every permutation and keep track of minimum cost permutation.
- Return the permutation with minimum cost.

Following is Dynamic Programming Algorithm:

Let the given set of vertices be 1, 2, 3, 4,….n. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values.

To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i. We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on.

Note :1 must be present in every subset.

If size of S is 2, then S must be 1, i, C(S, i) = dist(1, i) Else if size of S is greater than 2. C(S, i) = min C(S-i, j) + dis(j, i) where j belongs to S, j != i and j != 1.

For a set of size n, we consider n-2 subsets each of size n-1 such that all subsets don’t have nth in them. Using the above recurrence relation, we can write dynamic programming based solution. There are at most O(n*2n̂) subproblems, and each one takes linear time to solve. The total running time is therefore O(n2̂*2n̂). The time complexity is much less than O(n!), but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

III. Algorithm Analysis

Time complexity:

Naive Solution:- Time Complexity: (n!)

Dynamic Programming solution:- Since, there are at most O(n*2n̂) subproblems and each one takes linear time to solve. The total running time is therefore O(n2̂2n̂). The time complexity is much less than O(n!), but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

IV. Conclusion

TSP is a popular NP-Hard problem, but depending on the size of the input cities, it is possible to find an optimal or a near-optimal solution using various algorithms.

V. References

1.https://www.geeksforgeeks.org/dynamic-programming/

Appendix

Code for implementation of this paper is given below:

```cpp
#include <bits/stdc++.h>

using namespace std;

int tsp(const vector<vector<int>> &cities, int pos, int visited, vector<vector<int>> &state)
{
    if (visited == ((1 << cities.size()) - 1))
        return cities[pos][0]; // return to starting city

    if (state[pos][visited] != INT_MAX)
        return state[pos][visited];

    for (int i = 0; i < cities.size(); ++i)
    {
        // can't visit ourselves unless we're ending & skip if already visited
        if (i == pos || (visited & (1 << i)))
            continue;

        int distance = cities[pos][i] + tsp(cities, i, visited | (1 << i), state);
        if (distance < state[pos][visited])
            state[pos][visited] = distance;
    }

    return state[pos][visited];
}

int main()
{
    vector<vector<int>> cities = {{0, 10, 15, 20},
                                    {10, 0, 35, 25},
                                    {15, 35, 0, 30},
                                    {20, 25, 30, 0}};

    vector<vector<int>> state(cities.size());
    for (auto &neighbors : state)
        neighbors = vector<int>((1 << cities.size()) - 1, INT_MAX);

    cout << "minimum: " << tsp(cities, 0, 1, state) << endl;

    return 0;
}
```

Listing 1. Code for this paper