MINIMUM WEIGHT OF HAMILTONIAN CIRCUIT IN A COMPLETE GRAPH

A PROJECT REPORT

Submitted by

AYUSH RAJ [Reg No: RA2111028010007]

DHRUV ATRI [Reg No: RA2111028010068]

Under the Guidance of

DR. Kishore Anthuvan Sahayaraj K

Associate Professor, Computer Vision, Machine Learning

In partial fulfilment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

with a specialization in Cloud Computing

DEPARTMENT OF NETWORKING AND COMMUNICATIONS

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603 203

MAY 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this B.Tech project report titled "**MINIMUM WEIGHT OF HAMILTONIAN CIRCUIT IN COMPLETE GRAPH** " is the bonafide work of Mr. Dhruv Atri [Reg. No.: RA2111028010068] and Mr. Ayush Raj [Reg. No.RA211102801007] who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on thebasis of which a degree or award was conferred on an earlier occasion for this or any other candidate.

DR. Kishore Anthuvan Sahayaraj K

Associate Professor

DR. Annapurani Panaiyappan .K
HEAD OF THE DEPARTMENT

Department of Networking and Communication

SIGNATURE OF INTERNAL EXAMINER

SIGNATURE OF EXTERNAL EXAMINER

Department of Data Science and Business Systems

SRM Institute of Science and TechnologyOwn Work Declaration Form

Degree/ Course	**:** B.Tech in Computer Science and Engineering with specialization in Cloud Computing
Student Names	**:** Dhruv Atri , Ayush Raj
Registration Number	**:** RA2111028010068, RA211102801007
Title of Work.	**:** MINIMUM WEIGHT OF HAMILTONIAN CIRCUIT IN COMPLETE GRAPH

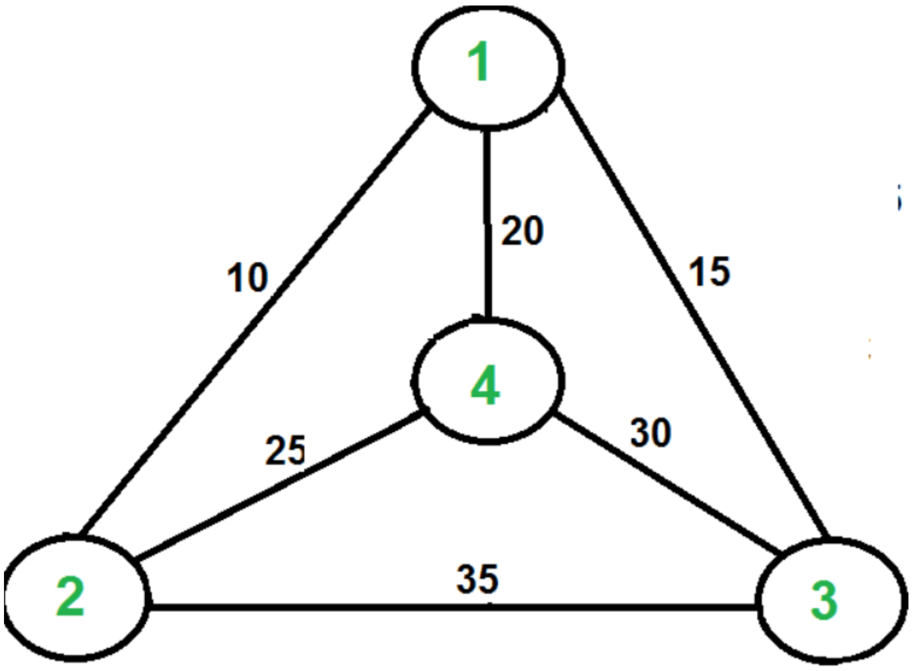
CONTENT

PROBLEM DEFINITION

To find minimum weight Hamiltonian Circuit in a complete graph.

In this problem travel represents edges , locations represent the vertex and the given graph is a complete graph , i.e. there exist travel from one location to other. Problem is Also Known as “TRAVELLING SALESMAN PROBLEM” .

Given a set of cities and the distance between every pair of cities, city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



PROBLEM EXPLANATION

We Need to find minimum weight Hamiltonian Circuit in a complete graph. In this problem travel represents edges , locations represent the vertex and the given graph is a complete graph , i.e. there exist travel from one location to other. We need to find the minimum weight of cycle such that each vertex is covered once and also salesman starts from one point , visits each city and return to that city.

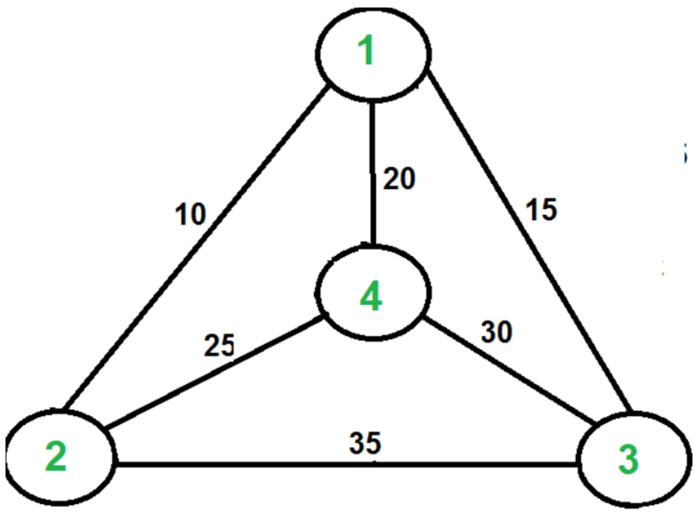
INPUT :

Graph with V vertex and E edges , and there exist a path between each vertex pair.

OUTPUT :

Cost of minimum Hamiltonian Circuit in Regular Graph.

EXAMPLE :



For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80. The problem is a famous NP-hard problem. There is no polynomial-time know solution for this problem. The following are different solutions for the traveling salesman problem.

DESIGN TECHNIQUES USED

The problem can be solved with multiple Design Techniques like :

1. Back Tracking
2. Branch & Bound
3. Dynamic Programming

As B&B and DP are optimised algorithms , Thus to solve this problem the following Approaches are used :

1. Branch&Bound
2. Dynamic Programming

1) BRANCH AND BOUND

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.

The Backtracking Solution can be optimized if we know a bound on best possible solution subtree rooted with every node. If the best in subtree is worse than current best, we can simply ignore this node and its subtrees. So we compute bound (best solution) for every node and compare the bound with current best solution before exploring the node.

2) DYNAMIC PROGRAMMING

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.

ALGORITHM FOR PROBLEM :

Using Branch&Bound :

Consider directed weighted graph G = (V, E, W), where node represents cities and weighted directed edges represents direction and distance between two cities.

1. Initially, graph is represented by cost matrix C, where

C_{ij} = cost of edge, if there is a direct path from city i to city j

C_{ij} = ∞, if there is no direct path from city i to city j.

2. Convert cost matrix to reduced matrix by subtracting minimum values from appropriate rows and columns, such that each row and column contains at least one zero entry.

3. Find cost of reduced matrix. Cost is given by summation of subtracted amount from the cost matrix to convert it in to reduce matrix.

4. Prepare state space tree for the reduce matrix

5. Find least cost valued node A (i.e. E-node), by computing reduced cost node matrix with every remaining node.

6. If <i, j> edge is to be included, then do following :

(a) Set all values in row i and all values in column j of A to ∞

(b) Set A[j, 1] = ∞

(c) Reduce A again, except rows and columns having all ∞ entries.

7. Compute the cost of newly created reduced matrix as,

Cost = L + Cost(i, j) + r

Where, L is cost of original reduced cost matrix and r is A[i, j].

8. If all nodes are not visited then go to step 4.

EXPLANATION FOR ALGORITHM :

As seen in the previous articles, in Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the subtree rooted with the node.
Note that the cost through a node includes two costs.
1) Cost of reaching the node from the root (When we reach a node, we have this cost computed)
2) Cost of reaching an answer from current node to a leaf (We compute a bound on this cost to decide whether to ignore subtree with this node or not).

Cost of a tour T = (1/2) * ∑ (Sum of cost of two edges
adjacent to u and in the
tour T)
where u ∈ V
For every vertex u, if we consider two edges through it in T, and sum their costs. The overall sum for all vertices would be twice of cost of tour T (We have considered every edge twice.)

(Sum of two tour edges adjacent to u) >= (sum of minimum weight
two edges adjacent to
u)

Cost of any tour >= 1/2) * ∑ (Sum of cost of two minimum
weight edges adjacent to u)
where u ∈ V

EXAMPLE:

For example, consider the above shown graph. Below are minimum cost two edges adjacent to every node.

Node	Least cost edges	Total cost
0	(0, 1), (0, 2)	25
1	(0, 1), (1, 3)	35
2	(0, 2), (2, 3)	45
3	(0, 3), (1, 3)	45

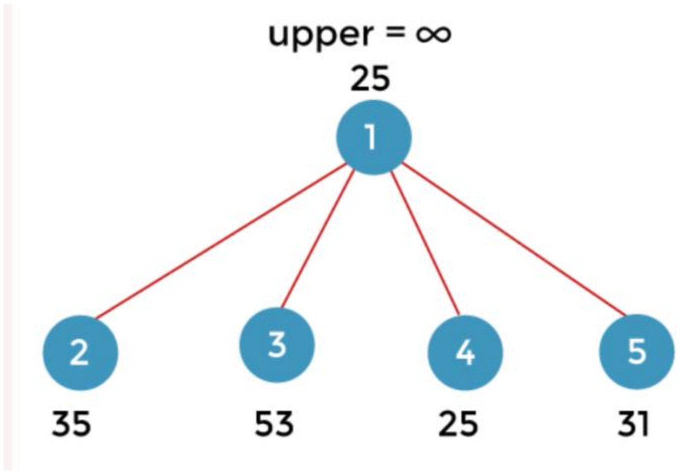
Thus a lower bound on the cost of any tour =

1/2(25 + 35 + 45 + 45)

= 75

Lower Bound for vertex 1 =

Old lower bound - ((minimum edge cost of 0 + minimum edge cost of 1) / 2) + (edge cost 0-1)



Dealing with other levels: As we move on to the next level, we again enumerate all possible vertices. For the above case going further after 1, we check out for 2, 3, 4, …n.
Consider lower bound for 2 as we moved from 1 to 1, we include the edge 1-2 to the tour and alter the new lower bound for this node.

Lower bound(2) = Old lower bound - ((second minimum edge cost of 1 + minimum edge cost of 2)/2) + edge cost 1-2)

It will Continue in Similar Fashion and ans will be :
The path of the tour would be 1->4->2->5->3.

CODE FOR GIVEN ALGORITHM (B&B) :

```
// C++ program to solve Traveling Salesman Problem
// using Branch and Bound.
#include <bits/stdc++.h>
using namespace std;
const int N = 4;
// final_path[] stores the final solution ie, the
// path of the salesman.
int final_path[N+1];

bool visited[N];

int final_res = INT_MAX;

// the final solution
void copyToFinal(int curr_path[])
{
    for (int i=0; i<N; i++)
        final_path[i] = curr_path[i];
    final_path[N] = curr_path[0];
}

// Function to find the minimum edge cost
// having an end at the vertex i
int firstMin(int adj[N][N], int i)
{
    int min = INT_MAX;
    for (int k=0; k<N; k++)
        if (adj[i][k]<min && i != k)
            min = adj[i][k];
    return min;
}

// having an end at the vertex i
int secondMin(int adj[N][N], int i)
{
    int first = INT_MAX, second = INT_MAX;
    for (int j=0; j<N; j++)
    {
        if (i == j)
            continue;

        if (adj[i][j] <= first)
        {
            second = first;
            first = adj[i][j];
        }
        else if (adj[i][j] <= second &&
                 adj[i][j] != first)
            second = adj[i][j];
    }
    return second;
}
void TSPRec(int adj[N][N], int curr_bound, int curr_weight,
            int level, int curr_path[])
```

```
{
    // base case is when we have reached level N which
    // means we have covered all the nodes once
    if (level==N)
    {
        // check if there is an edge from last vertex in
        // path back to the first vertex
        if (adj[curr_path[level-1]][curr_path[0]] != 0)
        {
            // curr_res has the total weight of the
            // solution we got
            int curr_res = curr_weight +
                    adj[curr_path[level-1]][curr_path[0]];

            // Update final result and final path if
            // current result is better.
            if (curr_res < final_res)
            {
                copyToFinal(curr_path);
                final_res = curr_res;
            }
        }
        return;
    }

    // for any other level iterate for all vertices to
    for (int i=0; i<N; i++)
    {

        // already)
        if (adj[curr_path[level-1]][i] != 0 &&
            visited[i] == false)
        {
            int temp = curr_bound;
            curr_weight += adj[curr_path[level-1]][i];

            if (level==1)
              curr_bound -= ((firstMin(adj, curr_path[level-1]) +
                              firstMin(adj, i))/2);
            else
              curr_bound -= ((secondMin(adj, curr_path[level-1]) +
                              firstMin(adj, i))/2);


            if (curr_bound + curr_weight < final_res)
            {
                curr_path[level] = i;
                visited[i] = true;

                // call TSPRec for the next level
                TSPRec(adj, curr_bound, curr_weight, level+1,
                       curr_path);
            }


            curr_weight -= adj[curr_path[level-1]][i];
            curr_bound = temp;

            // Also reset the visited array
            memset(visited, false, sizeof(visited));
```

```
                for (int j=0; j<=level-1; j++)
                    visited[curr_path[j]] = true;
            }
        }
}

// This function sets up final_path[]
void TSP(int adj[N][N])
{
    int curr_path[N+1];

    int curr_bound = 0;
    memset(curr_path, -1, sizeof(curr_path));
    memset(visited, 0, sizeof(curr_path));

    // Compute initial bound
    for (int i=0; i<N; i++)
        curr_bound += (firstMin(adj, i) +
                       secondMin(adj, i));

    curr_bound = (curr_bound&1)? curr_bound/2 + 1 :
                                 curr_bound/2;

    visited[0] = true;
    curr_path[0] = 0;

    TSPRec(adj, curr_bound, 0, 1, curr_path);
}

// Driver code
int main()
{
    //Adjacency matrix for the given graph
    int adj[N][N] = { {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    TSP(adj);

    printf("Minimum cost : %d\n", final_res);
    printf("Path Taken : ");
    for (int i=0; i<=N; i++)
        printf("%d ", final_path[i]);

    return 0;
}
```

OUTPUT :

```
Minimum cost : 80
Path Taken : 0 1 3 2 0
```

DYNAMIC PROGRAMMING ALGORITHM:

1. Travelling salesman problem takes a graph G {V, E} as an input and declare another graph as the output (say G’) which will record the path the salesman is going to take from one node to another.
2. The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
3. The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
4. Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.
5. Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.
6. However, if the origin is mentioned in the given problem, then the solution must always start from that node only. Let us look at some example problems to understand this better.

CODE USING DP :

```
#include <iostream>

using namespace std;

// there are four nodes in example graph (graph is 1-based)
const int n = 4;
// give appropriate maximum to avoid overflow
const int MAX = 1000000;

// dist[i][j] represents shortest distance to go from i to j
// this matrix can be calculated for any given graph using
// all-pair shortest path algorithms
int dist[n + 1][n + 1] = {
    { 0, 0, 0, 0, 0 },    { 0, 0, 10, 15, 20 },
    { 0, 10, 0, 25, 25 }, { 0, 15, 25, 0, 30 },
    { 0, 20, 25, 30, 0 },
};

// memoization for top down recursion
int memo[n + 1][1 << (n + 1)];
int fun(int i, int mask)
{
    // base case
    // if only ith bit and 1st bit is set in our mask,
    // it implies we have visited all other nodes already
    if (mask == ((1 << i) | 3))
        return dist[1][i];
    // memoization
    if (memo[i][mask] != 0)
        return memo[i][mask];

    int res = MAX; // result of this sub-problem



    for (int j = 1; j <= n; j++)
        if ((mask & (1 << j)) && j != i && j != 1)
            res = std::min(res, fun(j, mask & (~(1 << i)))
                                    + dist[j][i]);
    return memo[i][mask] = res;
}

int main()
{
    int ans = MAX;
    for (int i = 1; i <= n; i++)
        // try to go from node 1 visiting all nodes in
        // between to i then return from i taking the
        // shortest route to 1
        ans = std::min(ans, fun(i, (1 << (n + 1)) - 1)
                                + dist[i][1]);

    printf("The cost of most efficient tour = %d", ans);

    return 0;
}
```

Output: -

The cost of most efficient tour = 80

COMPLEXITY ANALYSIS :

1. Branch And Bound

Time Complexity: The worst case complexity of Branch and Bound remains same as that of the Brute Force clearly because in worst case, we may never get a chance to prune a node. Whereas, in practice it performs very well depending on the different instance of the TSP. The complexity also depends on the choice of the bounding function as they are the ones deciding how many nodes to be pruned.

Time Complexity: O(N2*log2N)

Auxiliary Space: O(N)

2. Dynamic Programming

Time Complexity : O(n2*2n) where O(n* 2n) are maximum number of unique subproblems/states and O(n) for transition (through for loop as in code) in every states.

Auxiliary Space: O(n*2n), where n is number of Nodes/Cities here.

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 a dynamic programming-based solution. There are at most O(n*2n) subproblems, and each one takes linear time to solve. The total running time is therefore O(n2*2n). The time complexity is much less than O(n!) but still exponential. The space required is also exponential. So this approach is also infeasible even for a slightly higher number of vertices. We will soon be discussing approximate algorithms for the traveling salesman problem.

CONCLUSION

Thus Though there are multiple approaches to the given problem of finding minimum weight Hamiltonian circuit. Thus the optimal solution for given problem is using branch and bound whose time complexity is 2n^2 in comparison to backtracking which takes 2^n exponential time and is an NP-Hard Algorithm. Thus we can find the path through which we can find minimum length Hamiltonian circuit also known as Traveling Salesman Problem.

REFERENCES

1. Geeksforgeeks.com
2. Introduction to Algorithms – Thomas H Corman
3. Data Structures and Algorithms Made Easy – Narasimha Karumanchi
4. www.javatpoint.com