IIT INDORE

28/04/2019

PROJECT REPORT

Approximation Analysis on Travelling Salesman Problem (TSP)

Submitted by-

1) Seemandhar Jain
   (170001046)

2) Akshit Khatgarh

   (170001004)

Submitted To-

Dr. Kapil Ahuja
Associate Professor in C.S.E.

IIT INDORE

I. INTRODUCTION 3

II. ALGORITHM ANALYSIS 7

III. ALGORITHM DESIGN 9

IV. IMPLEMENTATION AND RESULTS 13

V. CONCLUSION 17

VI. REFERENCES 18

Introduction-

There are various problems in computer science for which no polynomial time algorithm is known till now. These special problems are categorized as NP (not polynomial time) problems. A problem P is NP-hard if a polynomial-time algorithm for P would imply a polynomial-time algorithm for every problem in NP. Many problems of practical significance are NP-complete, yet they are too important to abandon merely because we don't know how to find an optimal solution in polynomial time. Even if a problem is NP-complete, there may be hope. We have at least three ways to get around NP-completeness.

• First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.

• Second, we may be able to isolate important special cases that we can solve in polynomial time.

• Third, we might come up with approaches to find near-optimal solutions in polynomial time (either in the worst case or the expected case). In practice, near-optimality is often good enough. We call an algorithm that returns near-optimal solutions an **approximation algorithm**. This project is all about development of polynomial-time approximation algorithms for several NP-complete problems.

In other words,

An **approximate algorithm** is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time.

Introduction to Approximation Algorithms

There are several optimization problems such as Minimum Spanning Tree (MST), Min-Cut, MaximumMatching, in which you can solve this exactly and efficiently in polynomial time. But many practical significant optimization problems are NP-Hard, in which we are unlikely to find an algorithm that solve the problem exactly in polynomial time. Examples of the standard NP-Hard problems with some of their brief description are as following:

• Traveling Salesman Problem (TSP) - finding a minimum cost tour of all cities

• Vertex Cover - find minimum set of vertex that covers all the edges in the graph (we will describe this in more detail)

These are NP-Hard problems, i.e., If we could solve any of these problems in polynomial time, then P = NP. An example of problem that is not known to be either NP-Hard: Given 2 graphs of n vertices, are they the same up to permutation of vertices? This is called Graph Isomorphism. As of now, there is no known polynomial exact algorithm for NP-Hard problems. However, it may be possible to find a near-optimal solutions in polynomial time. An algorithm that runs in polynomial time and outputs a solution close to the optimal solution is called an approximation algorithm.

Definition: Let P be a minimization problem, and I be an instance of P. Let A be an algorithm that finds feasible solution to instances of P. Let A(I) is the cost of the solution returned by A for instance I, and OP T(I) is the cost of the optimal solution (mimimum) for I. Then, A is said to be an α-approximation algorithm for P if

$\forall I, \quad A(I)/OPT(I) \leq \alpha$ where $\alpha \geq 1$.

Travelling salesman problem-

The Traveling Salesman Problem, often abbreviated TSP. The “TSP” problem is perhaps one of the most famous (and most studied) problems in combinatorial optimization.

Problem Definition

Given a set of cities (i.e., points), the goal of the traveling salesman problem is to find a minimum cost circuit that visits all the points. More formally, the problem is stated as follows:

Definition: Given a set V of n points and a distance function $d : V \times V \rightarrow R$, find a cycle C of minimum cost that contains all the points in V . The cost of a cycle $C = (e1, e2, \ldots, em)$ is defined to be $P\ e \in C\ d(e)$, and we assume that the distance function is non-negative $(i.e., d(x, y) \geq 0)$.

In the traveling-salesman problem, we are given a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge (u, v) ε E, and we must find a hamiltonian cycle (a tour) of G with minimum cost. TSP is an NP-complete problem. We formalize this notion by saying that the cost function c satisfies the triangle inequality if, for all vertices u, v, w ε $V, C(u, v) <= c(u, w) + c(w, v)$. The traveling-salesman problem is NP-complete even if we require that the cost function satisfy the triangle inequality. Thus, we should not expect to find a polynomial-time algorithm for solving this problem exactly. Instead, we look for good approximation algorithms.

The traveling salesman problem is NP-complete.

Proof:

First, we have to prove that TSP belongs to NP. If we want to check a tour for credibility, we check that the tour contains each vertex once. Then we sum the total cost of the edges and finally we check if the cost is minimum. This can be completed in polynomial time thus TSP belongs to NP.

Secondly we prove that TSP is NP-hard. One way to prove this is to show that Hamiltonian cycle TSP (given that the Hamiltonian cycle problem is NP-complete). Assume $G = (V, E)$ to be an instance of Hamiltonian cycle. An instance of TSP is then constructed. We create the complete $graph = (V, \leq P\ G'\ E')$, where E′ = {(i, j):i, j ∈ V and i ≠ j. Thus, the cost function is defined as

$$t(i,,j) = \begin{cases} 0 \text{ if } (\mathrm{i}, \mathrm{j}) \in E, \\ 1 \text{ if } (\mathrm{i}, \mathrm{j}) \notin E. \end{cases}$$

Now suppose that a Hamiltonian cycle h exists in G. It is clear that the cost of each edge in h is 0 in G′ as each edge belongs to E. Therefore, h has a cost of 0 in G′ . Thus, if graph G has a Hamiltonian cycle then graph G′ has a tour of 0 cost.

Conversely, we assume that G' has a tour h' of cost at most 0. The cost of edges in E' are 0 and 1 by definition. So each edge must have a cost of 0 as the cost of h' is 0. We conclude that h' contains only edges in E. So we have proven that G has a Hamiltonian cycle if and only if G' has a tour of cost at most 0. Thus TSP is **NP-complete**.

Algorithm Analysis

1)Naive Solution:

1) Consider city 1 as the starting and ending point.

2) Generate all (n-1)! of cities.

3) Calculate cost of every permutation and keep track of minimum cost permutation.

4) Return the permutation with minimum cost.

Time Complexity: $\boldsymbol{\Theta}(\boldsymbol{n}!)$

2)Dynamic Programming:

Let the given set of vertices be $\{1, 2, 3, 4, \ldots. n\}$. 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 that 1 must be present in every subset.

If size of S is 2, then S must be {1, i},

$$\boldsymbol{C(S,i) = dist(1,i)}$$

Else if size of S is greater than 2.

$\boldsymbol{C(S,i) = min\{ C(S-\{i\},j) + dis(j,i)\}}$ **where j belongs to** $\boldsymbol{S, j != i \; and \; j != 1}$.

Using the above recurrence relation, we can write dynamic programming based solution. There are at most $O(n*2^n)$ subproblems, and each one takes linear time to solve. The total running time is therefore $\boldsymbol{O(n\text{^}2 * 2\text{^}n)}$. 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.

3)Approximation algorithm-

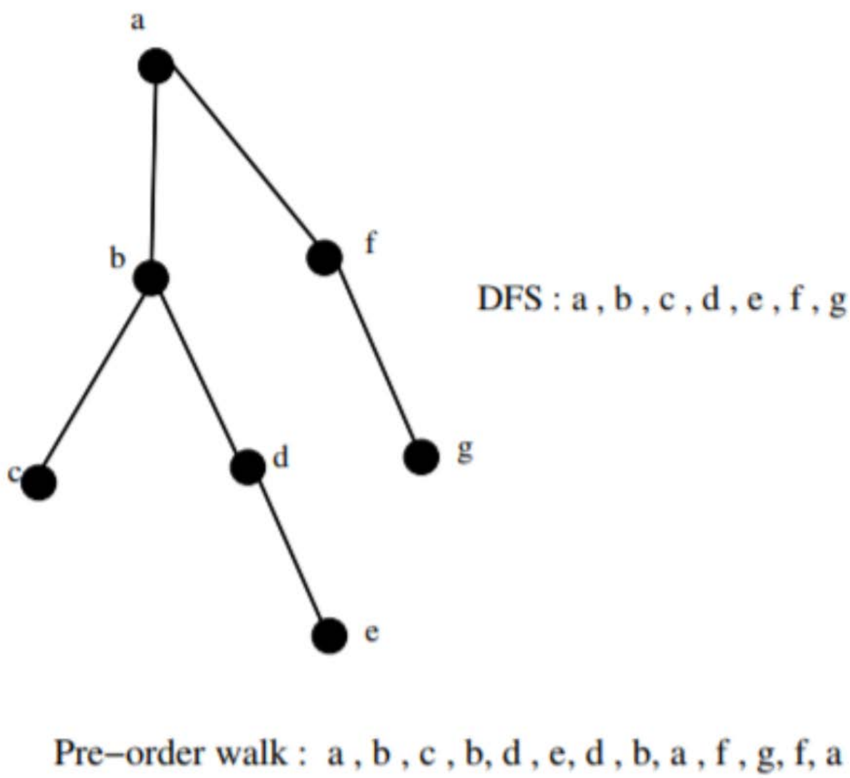
Given complete, undirected graph $G = (V,E)$ with non-negative integer cost $c(u,v)$ for each edge, find cheapest Hamiltonian cycle of G.

Consider two cases: with and without triangle inequality. c satisfies triangle inequality, if it is always cheapest to go directly from some u to some w; going by way of intermediate vertices can't be less expensive. Finding an optimal solution is NP-complete in both cases.

Suppose C is a cheapest hamiltonian cycle (tour). By removing one edge from C we obtain a path and this path is a spanning tree for G. Therefore the cost of C minus an edge is more than the cost of a minimum spanning tree (MST). Let T be a spanning tree of G then pre-order walk of T is a sequence of the vertices of T in DFS and its return.

Algorithm design

1) 2 Approximation pseudo algorithm-



We compute a minimum spanning tree whose weight is lower bound for length of optimal TSP tour. We use function $MSTPrim(G, c, r)$, which computes an MST for G and weight function c, given some arbitrary root r.

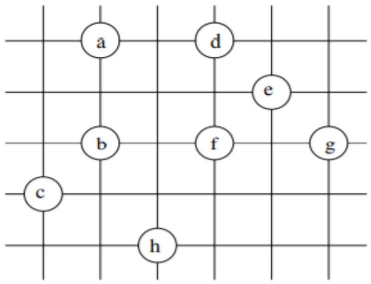
Approx-TSP-Tour ($G = (V, E), c : E \rightarrow R$)

1. Select arbitrary r ∈ V to be ”root”

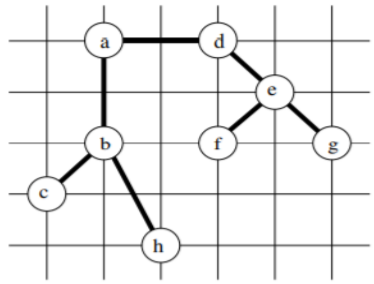
2. Compute MST T for G and c from root r using $MSTPrim(G, c, r)$

3. Let L be list of vertices visited in pre-order tree walk of T

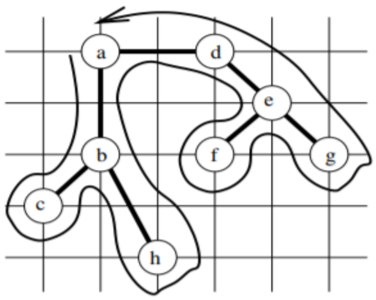
4. Return the Hamiltonian cycle that visits the vertices in the order L



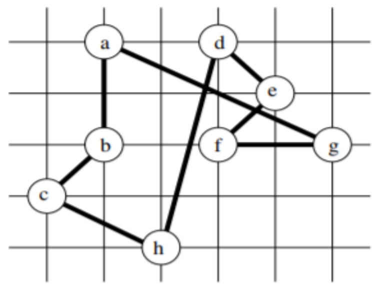
Set of points, lie in grid



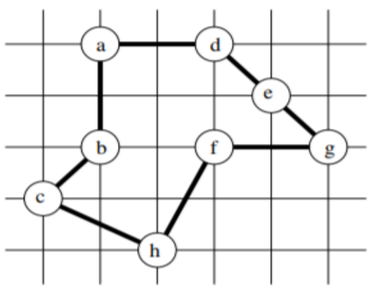
MST, root a



Pre–order walk



Resulting tour, cost ca. 19.1



Optimal tour, cost ca. 14.7

Theorem - Approx-TSP-Tour is a polynomial time 2-approximation algorithm for the TSP problem with triangle inequality.

Proof-

Polynomial running time obvious, simple MSTPrim takes $\theta(|V|^2)$, computing pre-order walk takes no longer.

Correctness obvious, pre-order walk is always a tour.

Let H $*$ denote an optimal tour for given set of vertices. Deleting any edge from $H *$ gives a spanning tree.

Thus, weight of minimum spanning tree is lower bound on cost of optimal tour: $c(T) \leq c(H *)$

A full walk of T lists vertices when they are first visited, and also when they are returned to, after visiting a subtree.

Example: $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$

Full walk W traverses every edge exactly twice, thus $c(W) = 2c(T)$

Together with $c(T) \leq c(H*)$, this gives $c(W) = 2c(T) \leq 2c(H*)$

Find a connection between cost of W and cost of "our" tour.

Problem: W is in general not a proper tour, since vertices may be visited more than once.

But: using the triangle inequality, we can delete a visit to any vertex from W and cost does not increase.

Deleting a vertex v from walk W between visits to u and w means going from u directly to w, without visiting v. We can consecutively remove all multiple visits to any vertex.

Example: full walk $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$ $becomes$ a, b, c, h, d, e, f, g.

This ordering (with multiple visits deleted) is identical to that obtained by pre-order walk of T (with each vertex visited only once). It certainly is a Hamiltonian cycle. Let's call it H.

H is just what is computed by Approx-TSP-Tour. H is obtained by deleting vertices from W ,

thus $c(H) \leq c(W)$

Conclusion: $\boldsymbol{c(H) \leq c(W) \leq 2c(H*)}$

2) 3/2-Approximation for Metric TSP

$C \leq 1.5 \times OPT$ (Christofides' Algorithm)

Two facts we need:

1. Minimum-weight matching in a weighted complete graph can be found in polynomial time.
2. A graph has a Eulerian tour if and only if all vertex degrees are even. In such a graph we can construct this tour in polynomial time. (A Eulerian path visits each edge exactly once and completes a circuit; a Eulerian tour is a Eulerian path that is also a tour, that is, it starts and ends in the same place.)

Algorithm:

1. Create an MST, as before.

Claim - In any graph, the number of vertices of odd degree must be even.

2. Find a minimum-weight perfect matching M* in the original graph between the vertices that have odd degree IN THE MST.

Claim - $M* \leq 0.5 \times OP\ T$

Proof –

Any tour can be decomposed into two matchings, M1 and M2, by alternating matched and unmatched edges. Therefore, $OPT = M1 + M2 \geq M * +M*$ Consider the subgraph $MST + M*$: Every vertex in this subgraph has even degree, so there exists some Eulerian tour E of this subgraph with cost exactly equal to $MST + M*$ since it uses each edge exactly once.

Therefore we have: $MST + M* \leq 0.5 \times OPT + OPT \leq 1.5 \times OPT$

Implementation

1) Naïve solution-

```cpp
int travllingSalesmanProblem(int graph[][V], int s)
{
    // store all vertex apart from source vertex
    vector<int> vertex;
    for (int i = 0; i < V; i++)
        if (i != s)
            vertex.push_back(i);

    // store minimum weight Hamiltonian Cycle.
    int min_path = INT_MAX;
    do {

        // store current Path weight(cost)
        int current_pathweight = 0;

        // compute current path weight
        int k = s;
        for (int i = 0; i < vertex.size(); i++) {
            current_pathweight += graph[k][vertex[i]];
            k = vertex[i];
        }
        current_pathweight += graph[k][s];

        // update minimum
        min_path = min(min_path, current_pathweight);

    } while (next_permutation(vertex.begin(), vertex.end()));

    return min_path;
}
```

Input-

```
{ { 0, 10, 15, 20 },
{ 10, 0, 35, 25 },
{ 15, 35, 0, 30 },
{ 20, 25, 30, 0 } };
```

Output- **80**

2) Dynamic Programming Solution-

```
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];
}
```

Input-

```
{ { 0, 10, 15, 20 },
{ 10, 0, 35, 25 },
{ 15, 35, 0, 30 },
{ 20, 25, 30, 0 } };
```

Output- **80**

3) 1.5 approximation code-

```
import argparse
from math import sqrt
import itertools
import sys

parser = argparse.ArgumentParser()
parser.add_argument('-f', action='store', dest='file_name', default='input.txt', required=False, help="input file location")

def read_from_file(file):
    lines = open(file).readlines()
    return [line[1:].strip().split(' ') for line in lines]

def write_to_file(file, line):
    with open(file + ".tour", "a") as output:
        output.write(" ".join(line))
        output.write('\n')

def calculate_distance(p1, p2):
    return sqrt((int(p2[0]) - int(p1[0])) ** 2 + (int(p2[1]) - int(p1[1])) ** 2)

def generate_distance_matrix(coordinates):
    matrix = []
    for a in coordinates:
        row = []
        for b in coordinates:
            row.append(calculate_distance(a,b))
        matrix.append(row)
    return matrix

class MST():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    # A utility function to print the constructed MST stored in parent[]
    def returnMST(self, parent):
```

```
    def returnMST(self, parent):
        MST = []
        for i in range(1,self.V):
            edge = (parent[i], i, self.graph[i][parent[i]])
            MST.append(edge)
            # print parent[i],"-",i,"\t",self.graph[i][ parent[i] ]
        return MST

    # A utility function to find the vertex with minimum distance value, from
    # the set of vertices not yet included in shortest path tree
    def minKey(self, key, mstSet):

        # Initilaize min value
        min = sys.maxint

        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v

        return min_index

    # Function to construct and print MST for a graph represented using
    # adjacency matrix representation
    def primMST(self):

        #Key values used to pick minimum weight edge in cut
        key = [sys.maxint] * self.V
        parent = [None] * self.V # Array to store constructed MST
        key[0] = 0   # Make key 0 so that this vertex is picked as first vertex
        mstSet = [False] * self.V

        parent[0] = -1  # First node is always the root of

        for cout in range(self.V):

```

```python

        for cout in range(self.V):

            u = self.minKey(key, mstSet)

            mstSet[u] = True
            for v in range(self.V):

                if self.graph[u][v] > 0 and mstSet[v] == False and\
                   key[v] > self.graph[u][v]:
                        key[v] = self.graph[u][v]
                        parent[v] = u

        return self.returnMST(parent)


def _odd_vertices_of_MST(M, number_of_nodes):
    """Returns the vertices having Odd degree in the Minimum Spanning Tree(MST).
    """
    odd_vertices = [0 for i in range(number_of_nodes)]
    for u,v,d in M:
        odd_vertices[u] = odd_vertices[u] + 1
        odd_vertices[v] = odd_vertices[v] + 1
    odd_vertices = [vertex for vertex, degree in enumerate(odd_vertices) if degree % 2 == 1]
    return odd_vertices

def bipartite_Graph(M, bipartite_set, odd_vertices):
    """
    """
    bipartite_graphs = []
    vertex_sets = []
    for vertex_set1 in bipartite_set:
        vertex_set1 = list(sorted(vertex_set1))
        vertex_set2 = []
        for vertex in odd_vertices:
            if vertex not in vertex_set1:
                vertex_set2.append(vertex)
    for vertex_set1 in bipartite_set:
        vertex_set1 = list(sorted(vertex_set1))
        vertex_set2 = []
        for vertex in odd_vertices:
            if vertex not in vertex_set1:
                vertex_set2.append(vertex)
        matrix = [[-1000000 for j in range(len(vertex_set2))] for i in range(len(vertex_set1))]
        for i in range(len(vertex_set1)):
            for j in range(len(vertex_set2)):
                if vertex_set1[i] < vertex_set2[j]:
                    matrix[i][j] = M[vertex_set1[i]][vertex_set2[j]]
                else:
                    matrix[i][j] = M[vertex_set2[j]][vertex_set1[i]]
        bipartite_graphs.append(matrix)
        vertex_sets.append([vertex_set1,vertex_set2])
    return [bipartite_graphs, vertex_sets]

def main():
    user_args = parser.parse_args()
    array_of_lines = read_from_file(user_args.file_name)
    mst = MST(len(array_of_lines))
    mst.graph = generate_distance_matrix(array_of_lines)
    triples = mst.primMST()
    print triples
    odd_vertices = _odd_vertices_of_MST(triples, len(array_of_lines))
    print odd_vertices
    bipartite_set = [set(i) for i in itertools.combinations(set(odd_vertices), len(odd_vertices)/2)]
    print bipartite_set
    bipartite_graphs = bipartite_Graph(mst.graph, bipartite_set, odd_vertices)
    print bipartite_graphs
    # print held_karp(generate_distance_matrix(array_of_lines))


if __name__ == "__main__":
    main()
```

Input-

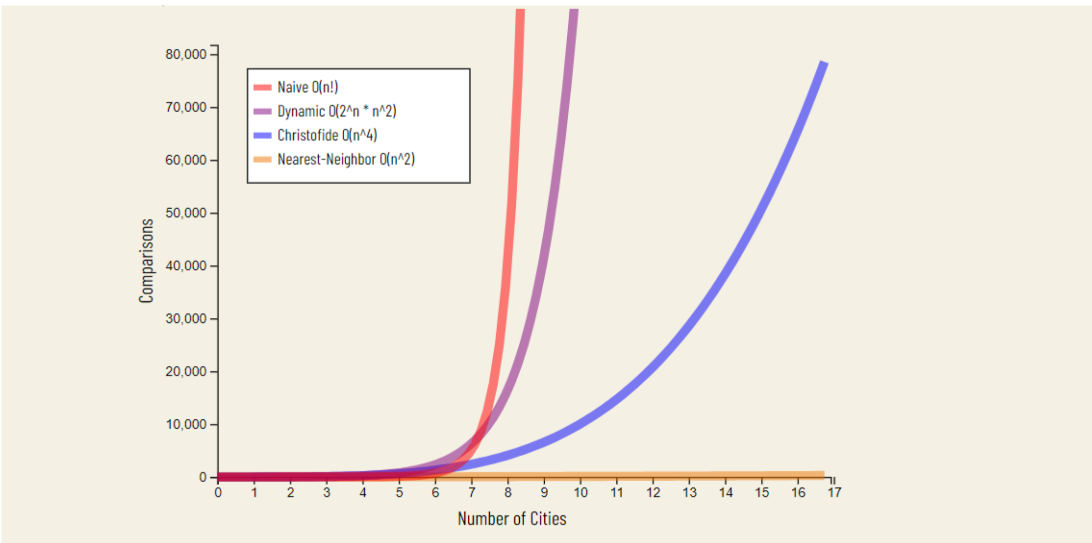
```
{ { 0, 10, 15, 20 },
{ 10, 0, 35, 25 },
{ 15, 35, 0, 30 },
{ 20, 25, 30, 0 } };
```

Output- 90(appx)

Conclusion-

1) Solution of TSP using D.P. gives the complexity of O(2^n*n^2).
2) Hamiltonian Cycle is polynomial time reducible to TSP which is NP-COMPLETE => TSP is NP-COMPLETE.
3) Using MST and triangular inequality ,we have shown an approximation algorithm which is accurate within 2-times of accurate result.
4) Since MST runs in order of polynomial time => our approximate algorithm also has a polynomial time complexity.
5) This approximation can further be reduce to 1.5-times accuracy with the help of MST, Perfect Matching algorithm and Eulers Theorem.
6) Number of comparisons required in

   order 2-approximation < Dynamic solution < naïve approach.



References:

1) A. Agrawal, P. Klein, and R. Ravi. When trees collide: An approximation algorithm for the generalized Steiner problem on networks. SIAM Journal on Computing, 24:440–456, 1995.

2) H. An, R. Kleinberg, and D. Shmoys. Approximation algorithms for the bottleneck asymmetric traveling salesman problem. In Proceedings of APPROX, pages 1–11, 2010.

3) A. Asadpour, M. X. Goemans, A. Madry, S. Oveis Gharan, and A. Saberi. An O(log n/ log log n)-approximation algorithm for the asymmetric traveling salesman problem. In Proceedings of SODA,2010. 4) H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein Introduction to Algorithms by Thomas. (Jul 31, 2009) 5) The Traveling Salesman Problem – Project Details CSE4080 (F06) Link to PDF

6) D. AppleGate, R. Bixby, V. Chvatal and W. Cook. On the Solution of the Traveling Salesman Problems. Documenta Mathematica – Extra Volume ICM, chapter 3, pp. 645-656, 1998.

7) S. Arora. The Approximability of NP-hard Problems. Princeton University, 1998.

8) S. Arora. Polynomial Time Approximation Schemes for Euclidean Traveling Saleman and Other Geometric Problems. Princeton University, 1996.