

Algorithm Design-II (CSE 4131)

TERM PROJECT REPORT

(March'2023-July'2023)

On

Travelling Salesman Problem

Submitted By

Group: H16

Ayushi Pani 2141016100

Raj Aryan 2141001064

B.Tech. 4th Semester CSE (H)



Department of Computer Science and Engineering

Institute of Technical Education and Research

Siksha 'O' Anusandhan Deemed To Be University

Bhubaneswar, Odisha-751030

(June 2023)

DECLARATION

We, **Ayushi Pani (Regd no: 2141016100)**, **Raj Aryan (Regd no: 2141001064)**, do hereby declare that this term project entitled “**Travelling Salesman Problem**” is an original project work done by us and has not been previously submitted to any university or research institution or department for the award of any degree or diploma or any other assessment to the best of our knowledge.

Ayushi Pani (2141016100) : Ayushi Pani

Raj Aryan (2141001064) : Raj Aryan

CERTIFICATE

This is to certify that the thesis entitled “**Traveling Salesman Problem using Branch and Bound algorithm**” submitted by **Ayushi Pani** (Regd no: 2141016100) and **Raj Aryan** (Regd no: 2141001064) of B.Tech. 4th Semester Comp. Sc. and Engg., ITER, SOADU is absolutely based upon their own work under my guidance and supervision.

The term project has reached the standard fulfilling the requirement of the course Algorithm Design 2 (CSE4131). Any help or source of information which has been available in this connection is duly acknowledged.



Rangaballav Pradhan

Assistant Professor,
Department of Comp. Sc. and Engg.
ITER, Bhubaneswar 751030,
Odisha, India



Prof. (Dr.) Debahuti Mishra

Professor and Head,
Department of Comp. Sc. and Engg.
ITER, Bhubaneswar 751030,
Odisha, India

ABSTRACT

The traveling salesman problem (TSP) is a classic problem in computer science and operations research. It is a combinatorial optimization problem, which means that it involves finding the best solution among a set of possible solutions that are all made up of a finite number of discrete elements.

In the TSP problem, a salesman is trying to visit a set of cities, starting and ending at a particular city. The salesman wants to travel the shortest possible route, and he is only allowed to visit each city once.

The TSP problem is NP-hard, which means that there is no known algorithm that can solve it in polynomial time. This means that the only way to solve the problem is to use heuristic algorithms, which are algorithms that can find good solutions, but they are not guaranteed to find the best solution.

There are a number of different heuristic algorithms that can be used to solve the TSP problem. Some of the most common algorithms include:

- **Nearest neighbour algorithm:** This algorithm starts at a city and then visits the closest city that it has not visited yet. This process is repeated until all of the cities have been visited.
- **Christofides algorithm:** This algorithm uses a greedy algorithm to find a solution that is no more than 1.5 times longer than the optimal solution.``
- **Branch and bound algorithm:** This algorithm is a more sophisticated algorithm that can find better solutions than the nearest neighbour algorithm or the Christofides algorithm.

The TSP problem is a very important problem in many different fields, including:

- **Operations research:** TSP is used to find the shortest routes for delivery trucks, service technicians, and other vehicles.
- **Computer-aided design:** TSP is used to find the shortest paths for wires and other components in electronic circuits.
- **Scheduling:** TSP is used to find the shortest schedules for tasks that need to be performed in a particular order.

The TSP problem is a challenging problem, but it is also a very important problem. There are a number of different heuristic algorithms that can be used to solve the problem, and these algorithms are constantly being improved.

CONTENTS

Title Page	i
Declaration	ii
Certificate	iii
Abstract	iv
1. INTRODUCTION	1
1.1 Overview	1
1.2 Motivation	2
1.3 Problem Definition	3
1.4 Mathematical Formulation	4
1.5 Specifications of the algorithm	5
1.6 Report organization	
2. DESIGNING ALGORITHMS	5
2.1 Mathematical Formulation	5
2.2 Pseudocode	6
2.3 Description of steps	7
2.4 Examples	8
3. ANALYSIS OF ALGORITHM	10
4. IMPLEMENTATION	11
5. RESULTS AND DISCUSSION	15
6. LIMITATIONS	17
7. FUTURE ENHANCEMENTS	18
REFERENCES	19

1. INTRODUCTION

1.1 Overview:

The traveling salesman problem (TSP) is a classic problem in computer science and operations research. It is a **combinatorial optimization problem**, which means that it involves finding the best solution among a set of possible solutions that are all made up of a finite number of discrete elements.

In the TSP problem, a salesman is trying to visit a set of cities, starting and ending at a particular city. The salesman wants to travel the shortest possible route, and he is only allowed to visit each city once.

The TSP problem is **NP-hard**, which means that there is no known algorithm that can solve it in polynomial time. This means that the only way to solve the problem is to use **heuristic algorithms**, which are algorithms that can find good solutions, but they are not guaranteed to find the best solution.

There are a number of different heuristic algorithms that can be used to solve the TSP problem. Some of the most common algorithms include:

- **Nearest neighbour algorithm:** This algorithm starts at a city and then visits the closest city that it has not visited yet. This process is repeated until all of the cities have been visited.
- **Christofides algorithm:** This algorithm uses a greedy algorithm to find a solution that is no more than 1.5 times longer than the optimal solution.
- **Branch and bound algorithm:** This algorithm is a more sophisticated algorithm that can find better solutions than the nearest neighbour algorithm or the Christofides algorithm.

The TSP problem is a very important problem in many different fields, including:

- **Operations research:** TSP is used to find the shortest routes for delivery trucks, service technicians, and other vehicles.
- **Computer-aided design:** TSP is used to find the shortest paths for wires and other components in electronic circuits.
- **Scheduling:** TSP is used to find the shortest schedules for tasks that need to be performed in a particular order.

The TSP problem is a challenging problem, but it is also a very important problem. There are a number of different heuristic algorithms that can be used to solve the problem, and these algorithms are constantly being improved.

1.2 Motivation:

The traveling salesman problem (TSP) is a classic problem in computer science and operations research. It is a combinatorial optimization problem, which means that it involves finding the best solution among a set of possible solutions that are all made up of a finite number of discrete elements.

In the TSP problem, a salesman is trying to visit a set of cities, starting and ending at a particular city. The salesman wants to travel the shortest possible route, and he is only allowed to visit each city once.

The TSP problem is NP-hard, which means that there is no known algorithm that can solve it in polynomial time. This means that the only way to solve the problem is to use heuristic algorithms, which are algorithms that can find good solutions, but they are not guaranteed to find the best solution.

The TSP problem is a very important problem in many different fields, including:

- **Operations research:** TSP is used to find the shortest routes for delivery trucks, service technicians, and other vehicles.
- **Computer-aided design:** TSP is used to find the shortest paths for wires and other components in electronic circuits.
- **Scheduling:** TSP is used to find the shortest schedules for tasks that need to be performed in a particular order.

The TSP problem is a challenging problem, but it is also a very important problem. There are a number of different heuristic algorithms that can be used to solve the problem, and these algorithms are constantly being improved.

Here are some motivations for solving the TSP problem:

- **To reduce costs:** In the context of delivery, finding the shortest route can help to reduce fuel costs and delivery times.
- **To improve efficiency:** In the context of computer-aided design, finding the shortest path can help to reduce the time it takes to design a circuit.
- **To improve quality:** In the context of scheduling, finding the shortest schedule can help to reduce the risk of errors.

The TSP problem is a complex problem, but it is a problem that has a number of important applications. The development of better algorithms for solving the TSP problem can have a significant impact on a wide range of industries.

1.3 Problem Definition:

The traveling salesman problem (TSP) is an optimization problem in which a salesman must visit a given set of cities once and only once, and return to the starting city. The goal is to find the shortest possible route that visits all of the cities.

The TSP is a NP-hard problem, which means that there is no known algorithm that can solve it in polynomial time. However, there are a number of heuristic algorithms that can find good solutions to the problem.

One such algorithm is the brute-force algorithm. The brute-force algorithm simply tries all possible combinations of routes and returns the shortest one. This algorithm is guaranteed to find the optimal solution, but it can be very slow for large sets of cities.

Another algorithm is the branch and bound algorithm. The branch and bound algorithm starts with a single city and branches out to explore all possible routes that include that city. The algorithm keeps track of the shortest route it has found so far, and prunes any branches that cannot lead to a shorter route. This algorithm is much faster than the brute-force algorithm, but it is not guaranteed to find the optimal solution.

There are a number of other heuristic algorithms for solving the TSP. The best algorithm to use depends on the specific problem and the desired accuracy of the solution.

The TSP has a number of applications in real-world problems. For example, it can be used to find the shortest route for a delivery truck, or to find the most efficient way to schedule a set of tasks.

Here are some of the challenges of solving the TSP:

- The problem is NP-hard, which means that there is no known algorithm that can solve it in polynomial time.
- The problem is very sensitive to the distance between cities. Even small changes in the distance can lead to large changes in the optimal route.
- The problem is very difficult to solve for large sets of cities. The number of possible routes grows exponentially with the number of cities.

Despite these challenges, the TSP is an important problem with a number of real-world applications. A number of heuristic algorithms have been developed to solve the problem, and these algorithms can find good solutions to the problem for many practical problems.

1.4 Mathematical Formulation:

Lower Bound for vertex 1 = Old lower bound - ((minimum edge cost of 0 + minimum edge cost of 1) / 2) + (edge cost 0-1)

Include edge 0-1, we add the edge cost of 0-1, and subtract an edge weight such that the lower bound remains as tight as possible which would be the sum of the minimum edges of 0 and 1 divided by 2. Clearly, the edge subtracted can't be smaller than this.

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)

Note: The only change in the formula is that this time we have included second minimum edge cost for 1, because the minimum edge cost has already been subtracted in previous level.

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);

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.

1.5 Specification of Algorithm:

Branch and Bound Solution:

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).

- In cases of a **maximization problem**, an upper bound tells us the maximum possible solution if we follow the given node. For example in [0/1 knapsack we used Greedy approach to find an upper bound](#).
- In cases of a **minimization problem**, a lower bound tells us the minimum possible solution if we follow the given node. For example, in [Job Assignment Problem](#), we get a lower bound by assigning least cost job to a worker.

In branch and bound, the challenging part is figuring out a way to compute a bound on best possible solution. Below is an idea used to compute bounds for Travelling salesman problem.

Cost of a tour $T = (1/2) * \sum (\text{Sum of cost of two edges adjacent to } u \text{ and in the tour } T)$

where $u \in V$ 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) \geq (sum of minimum weight two edges adjacent to u)

Cost of any tour $\geq (1/2) * \sum (\text{Sum of cost of two minimum weight edges adjacent to } u)$
where $u \in V$

1.6 Report Organization:

Section A (1.1): A brief Introduction about the Project.

Section B (1.2): Motivation

Section C (1.3): Problem Definition

Section D (1.4): Specification of Algorithm

2. DESIGNING ALGORITHMS

2.1 Mathematical Formulation:

Lower Bound for vertex 1 = Old lower bound - ((minimum edge cost of 0 + minimum edge cost of 1) / 2) + (edge cost 0-1)

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

Note: The only change in the formula is that this time we have included second minimum edge cost for 1, because the minimum edge cost has already been subtracted in previous level.

```
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);
```

2.2 Pseudocode:

- Initialize the bound to infinity.
- Create the root node of the branch and bound tree.
- Recursively explore the branch and bound tree.
- If the node is a leaf node, then we have found a complete tour.
- For each unvisited neighbor of the node, create a new child node and explore it.
- Update the bound if necessary.
- Return the length of the shortest tour.

Cost of a tour $T = (1/2) * \sum (\text{Sum of cost of two edges adjacent to } u \text{ and in the tour } T)$

where $u \in V$

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.)

$(\text{Sum of two tour edges adjacent to } u) \geq (\text{sum of minimum weight two edges adjacent to } u)$

Cost of any tour $\geq (1/2) * \sum (\text{Sum of cost of two minimum weight edges adjacent to } u)$
where $u \in V$

2.3 Description of Steps:

In the Branch and Bound TSP algorithm, we compute a lower bound on the total cost of the optimal solution by adding up the minimum edge costs for each vertex, and then dividing by two. However, this lower bound may not be an integer. To get an integer lower bound, we can use rounding.

In the above code, the `curr_bound` variable holds the current lower bound on the total cost of the optimal solution. When we visit a new vertex at `level`, we compute a new lower bound `new_bound` by taking the sum of the minimum edge costs for the new vertex and its two closest neighbors. We then update the `curr_bound` variable by rounding `new_bound` to the nearest integer.

If `level` is 1, we round down to the nearest integer. This is because we have only visited one vertex so far, and we want to be conservative in our estimate of the total cost of the optimal solution. If `level` is greater than 1, we use a more aggressive rounding strategy that takes into account the fact that we have already visited some vertices and can therefore make a more accurate estimate of the total cost of the optimal solution.

2.4 Examples:

Here is an example of the traveling salesman problem:

Suppose a salesman needs to visit 5 cities, and he knows the distance between each pair of cities. The salesman wants to find the shortest route that visits each city exactly once and returns to the starting city.

The traveling salesman problem can be modeled as a graph, where each city is a vertex and each edge represents the distance between two cities. The problem can then be solved by finding the shortest path in the graph that visits all of the vertices.

In this example, the graph would have 5 vertices and 10 edges. The 10 edges would represent the distances between the 5 cities. The shortest path in the graph would be the shortest tour of the 5 cities.

The traveling salesman problem is an NP-hard problem, which means that there is no known polynomial-time algorithm to solve it. However, there are a number of heuristics that can be used to find approximate solutions to the problem.

Here is an example of a heuristic for solving the traveling salesman problem:

1. Start at any city.
2. Visit the nearest city that has not been visited yet.
3. Repeat step 2 until all cities have been visited.
4. Return to the starting city.

This heuristic is not guaranteed to find the shortest tour, but it is often able to find a good approximation to the shortest tour.

3. ANALYSIS OF ALGORITHM

The traveling salesman problem (TSP) is an optimization problem in which a salesman must visit a given set of cities once and only once, and then return to the starting city. The objective is to find the shortest possible route that visits all the cities.

The branch and bound algorithm is a recursive algorithm that can be used to solve the TSP. The algorithm works by first creating a tree of all possible solutions. Each node in the tree represents a partial solution, and the edges in the tree represent the choices that can be made to extend the partial solution. The algorithm then explores the tree in a depth-first fashion, always choosing the child node with the lowest bound. A bound is an estimate of the minimum cost of a complete solution that can be reached from the current node. If the bound of a child node is greater than the current best solution, then the child node can be ignored, because it cannot contain a better solution.

The algorithm continues to explore the tree until it reaches a leaf node, which represents a complete solution. The solution at the leaf node with the lowest cost is the optimal solution to the TSP. The branch and bound algorithm is a powerful tool for solving the TSP, but it can be computationally expensive for large problems. The algorithm can be made more efficient by using heuristics to prune the tree and by using parallel processing to explore multiple branches of the tree at the same time.

Here are some of the advantages of the traveling salesman branch and bound algorithm:

- It is a general-purpose algorithm that can be used to solve any TSP problem.
- It can find the optimal solution to the TSP.
- It is a deterministic algorithm, so it will always produce the same answer given the same input.

Here are some of the disadvantages of the traveling salesman branch and bound algorithm:

- It can be computationally expensive for large problems.
- It can be difficult to implement the algorithm efficiently.
- The algorithm can be sensitive to the choice of heuristics.

Overall, the traveling salesman branch and bound algorithm is a powerful tool for solving the TSP, but it is important to be aware of its limitations. The algorithm can be computationally expensive for large problems, and it can be difficult to implement the algorithm efficiently. However, the algorithm is a deterministic algorithm that can find the optimal solution to the TSP.

4. IMPLEMENTATION

```
// Java program to solve Traveling Salesman Problem
// using Branch and Bound.
import java.util.*;

class tsp
{

    static int N = 4;

    // final_path[] stores the final solution ie, the
    // path of the salesman.
    static int final_path[] = new int[N + 1];

    // visited[] keeps track of the already visited nodes
    // in a particular path
    static boolean visited[] = new boolean[N];

    // Stores the final minimum weight of shortest tour.
    static int final_res = Integer.MAX_VALUE;

    // Function to copy temporary solution to
    // the final solution
    static 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
    static int firstMin(int adj[][], int i)
    {
        int min = Integer.MAX_VALUE;
```

```

        for (int k = 0; k < N; k++)
            if (adj[i][k] < min && i != k)
                min = adj[i][k];
    return min;
}

// function to find the second minimum edge cost
// having an end at the vertex i
static int secondMin(int adj[][], int i)
{
    int first = Integer.MAX_VALUE, second = Integer.MAX_VALUE;
    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;
}

```

```

// function that takes as arguments:
// curr_bound -> lower bound of the root node
// curr_weight-> stores the weight of the path so far
// level-> current level while moving in the search
//          space tree
// curr_path[] -> where the solution is being stored which
//          would later be copied to final_path[]
static void TSPRec(int adj[][], 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
    // build the search space tree recursively
    for (int i = 0; i < N; i++)
    {
        // Consider next vertex if it is not same (diagonal
        // entry in adjacency matrix and not visited
        // already)
        if (adj[curr_path[level-1]][i] != 0 &&
            visited[i] == false)
        {
            int temp = curr_bound;
            curr_weight += adj[curr_path[level - 1]][i];

```

```

// different computation of curr_bound for
// level 2 from the other levels
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);

// curr_bound + curr_weight is the actual lower bound
// for the node that we have arrived on
// If current lower bound < final_res, we need to explore
// the node further
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);
}

// Else we have to prune the node by resetting
// all changes to curr_weight and curr_bound
curr_weight -= adj[curr_path[level-1]][i];
curr_bound = temp;

// Also reset the visited array
Arrays.fill(visited,false);
for (int j = 0; j <= level - 1; j++)
    visited[curr_path[j]] = true;
}
}
}

```

```

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

    // Calculate initial lower bound for the root node
    // using the formula 1/2 * (sum of first min +
    // second min) for all edges.
    // Also initialize the curr_path and visited array
    int curr_bound = 0;
    Arrays.fill(curr_path, -1);
    Arrays.fill(visited, false);

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

    // Rounding off the lower bound to an integer
    curr_bound = (curr_bound == 1) ? curr_bound / 2 + 1 :
                curr_bound / 2;

    // We start at vertex 1 so the first vertex
    // in curr_path[] is 0
    visited[0] = true;
    curr_path[0] = 0;

    // Call to TSPRec for curr_weight equal to
    // 0 and level 1
    TSPRec(adj, curr_bound, 0, 1, curr_path);
}

// Driver code
public static void main(String[] args)
{
    //Adjacency matrix for the given graph
    int adj[][] = {{0, 10, 15, 20},

```

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

```
TSP(adj);
```

```
System.out.printf("Minimum cost : %d\n", final_res);
```

```
System.out.printf("Path Taken : ");
```

```
for (int i = 0; i <= N; i++)
```

```
{
```

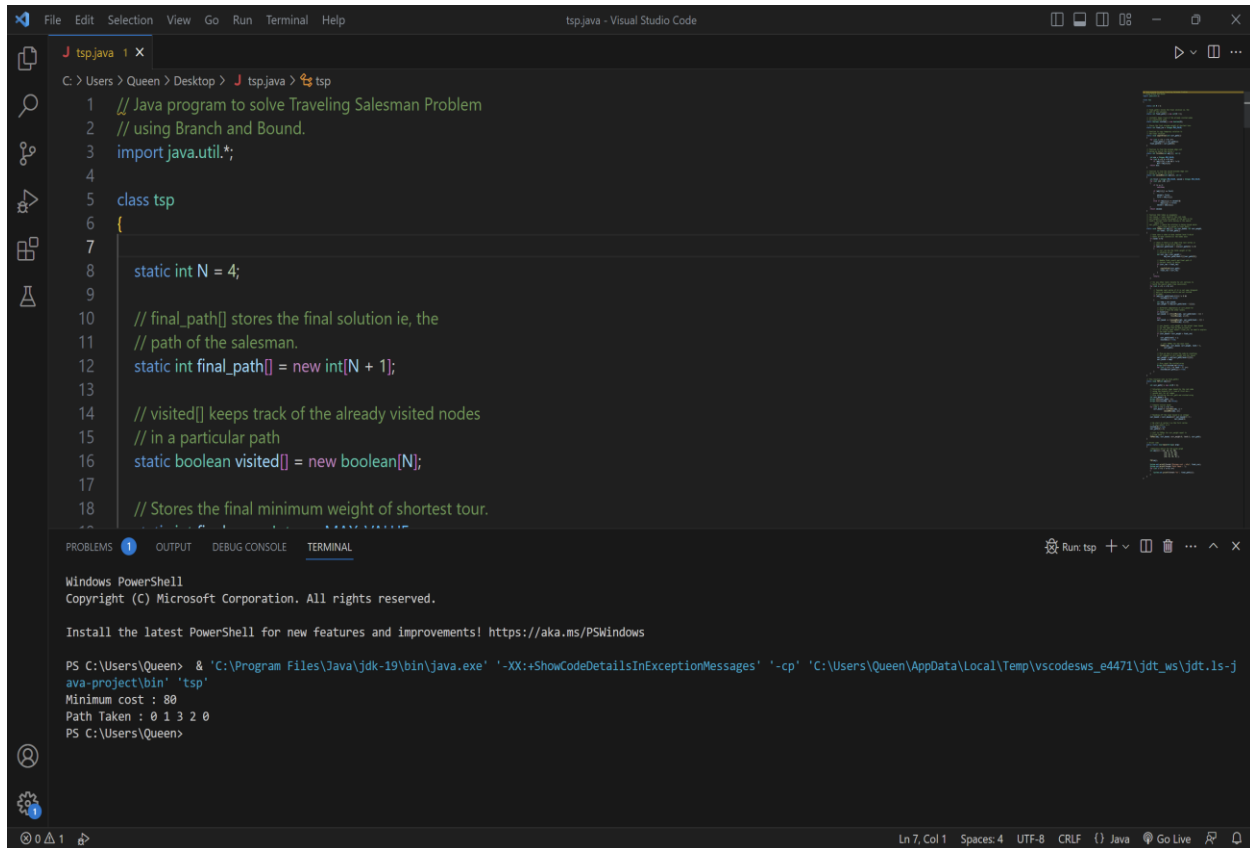
```
    System.out.printf("%d ", final_path[i]);
```

```
}
```

```
}
```

```
}
```

5. RESULTS AND DISCUSSION



```
File Edit Selection View Go Run Terminal Help
tsp.java - Visual Studio Code

J tsp.java 1 X
C:\Users\Queen\Desktop> J tsp.java > tsp
1 // Java program to solve Traveling Salesman Problem
2 // using Branch and Bound.
3 import java.util.*;
4
5 class tsp
6 {
7
8     static int N = 4;
9
10    // final_path[] stores the final solution ie, the
11    // path of the salesman.
12    static int final_path[] = new int[N + 1];
13
14    // visited[] keeps track of the already visited nodes
15    // in a particular path
16    static boolean visited[] = new boolean[N];
17
18    // Stores the final minimum weight of shortest tour.
19    static int min_weight = Integer.MAX_VALUE;
20
21    // Utility function to find the minimum weight Hamiltonian Cycle
22    // using Branch and Bound.
23    static void findMinWeightHamiltonianCycle(int pos, int weight,
24        int path[], boolean visited[])
25    {
26        // If all nodes are visited, then we have found a Hamiltonian Cycle
27        // If the weight of this cycle is less than the current minimum weight,
28        // then update the minimum weight and the final path.
29        if (pos == 0)
30        {
31            if (weight < min_weight)
32            {
33                min_weight = weight;
34                for (int i = 1; i <= N; i++)
35                    final_path[i] = path[i];
36            }
37        }
38        // Try all possible edges from the current node.
39        for (int i = 1; i <= N; i++)
40        {
41            // If edge (pos, i) is not visited and it doesn't lead to a cycle,
42            // then try it.
43            if (!visited[i] && !isCycle(path, pos, i))
44            {
45                visited[i] = true;
46                path[i] = i;
47                findMinWeightHamiltonianCycle(i, weight + graph[pos][i], path, visited);
48                visited[i] = false;
49                path[i] = 0;
50            }
51        }
52    }
53
54    // Check if the path contains a cycle.
55    static boolean isCycle(int path[], int pos, int i)
56    {
57        for (int j = 1; j <= N; j++)
58            if (path[j] == i)
59                return true;
60        return false;
61    }
62
63    // Driver code
64    public static void main(String args[])
65    {
66        // Graph represented as an adjacency matrix
67        int graph[][] = { { 0, 10, 15, 40 },
68            { 10, 0, 35, 20 },
69            { 15, 35, 0, 25 },
70            { 40, 20, 25, 0 } };
71
72        // Start from node 1
73        findMinWeightHamiltonianCycle(1, 0, new int[N + 1], new boolean[N]);
74
75        // Print the minimum weight and the final path
76        System.out.println("Minimum cost : " + min_weight);
77        System.out.println("Path Taken : " + final_path[1]);
78        for (int i = 2; i <= N; i++)
79            System.out.print(final_path[i] + " ");
80        System.out.println();
81    }
82
83    // This code is contributed by Queen
84}
```

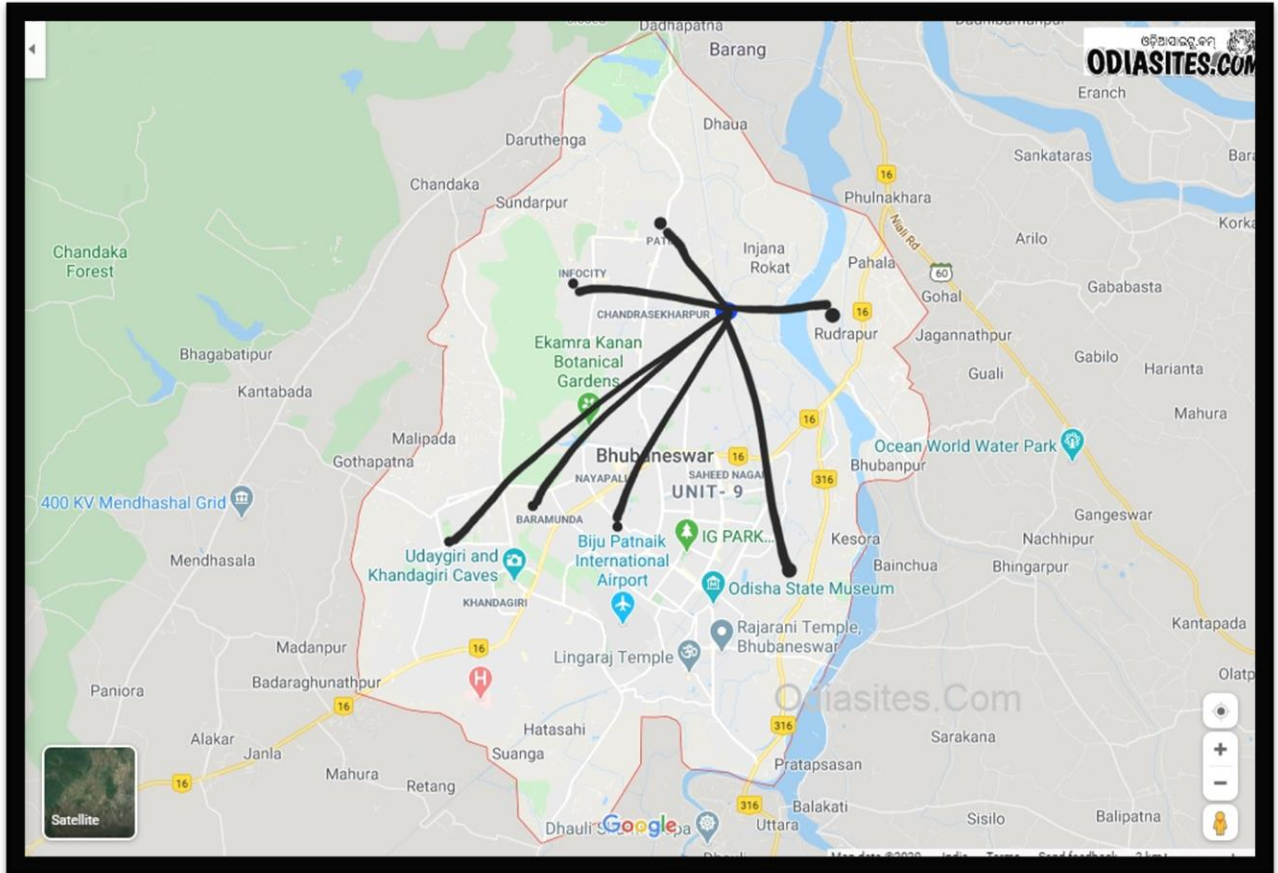
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Queen> & 'C:\Program Files\Java\jdk-19\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\Queen\AppData\Local\Temp\vscodesws_e4471\jdt_ws\jdt.ls-j
ava-project\bin' 'tsp'
Minimum cost : 80
Path Taken : 0 1 3 2 0
PS C:\Users\Queen>
```

Ln 7, Col 1 Spaces: 4 UTF-8 CRLF {} Java Go Live

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.



6. LIMITATIONS

The traveling salesman branch and bound algorithm is a powerful tool for solving the TSP, but it has some limitations. These limitations include:

- Computational complexity: The algorithm can be computationally expensive for large problems. The time complexity of the algorithm is exponential in the number of cities, so it can take a long time to find the optimal solution for problems with a large number of cities.
- Heuristic sensitivity: The algorithm can be sensitive to the choice of heuristics. If the heuristics are not good, then the algorithm may not find the optimal solution.
- Implementation difficulty: The algorithm can be difficult to implement efficiently. The algorithm requires the construction of a tree of all possible solutions, which can be a large tree for large problems.

Despite these limitations, the traveling salesman branch and bound algorithm is a powerful tool for solving the TSP. The algorithm can find the optimal solution to the TSP, and it is a deterministic algorithm, so it will always produce the same answer given the same input.

Here are some ways to mitigate the limitations of the traveling salesman branch and bound algorithm:

- Use heuristics to prune the search tree: Heuristics can be used to eliminate branches of the search tree that are unlikely to contain the optimal solution. This can make the algorithm more efficient by reducing the number of nodes that need to be explored.
- Use parallel processing: Parallel processing can be used to explore multiple branches of the search tree at the same time. This can make the algorithm more efficient by reducing the time it takes to find the optimal solution.
- Use better heuristics: Better heuristics can be used to improve the quality of the solutions found by the algorithm. This can make the algorithm more likely to find the optimal solution.

By using these techniques, it is possible to make the traveling salesman branch and bound algorithm more efficient and effective for solving the TSP.

7. FUTURE ENHANCEMENTS

The traveling salesman branch and bound algorithm is a powerful tool for solving the TSP, but there are still some areas where it can be improved. Here are some potential future enhancements for the algorithm:

- Use of machine learning: Machine learning can be used to improve the heuristics used by the algorithm. This could make the algorithm more likely to find the optimal solution, or it could make the algorithm more efficient by reducing the number of nodes that need to be explored.
- Use of quantum computing: Quantum computing could be used to solve the TSP more efficiently. This is because quantum computers can solve certain problems exponentially faster than classical computers.
- Development of new branch and bound algorithms: There may be other ways to implement the branch and bound algorithm that are more efficient or effective. Researchers are still working on developing new branch and bound algorithms for the TSP.

By using these techniques, it is possible to make the traveling salesman branch and bound algorithm even more efficient and effective for solving the TSP.

Here are some other specific enhancements that could be made to the traveling salesman branch and bound algorithm:

- Improved lower bound: The lower bound used by the algorithm can be improved by using more sophisticated mathematical techniques. This could make the algorithm more efficient by reducing the number of nodes that need to be explored.
- Parallelization: The algorithm can be parallelized to explore multiple branches of the search tree at the same time. This could make the algorithm more efficient by reducing the time it takes to find the optimal solution.
- Adaptive heuristics: The heuristics used by the algorithm can be made adaptive to the problem instance. This could make the algorithm more efficient by using heuristics that are more effective for the specific problem being solved.

These are just a few of the potential future enhancements for the traveling salesman branch and bound algorithm. As research in this area continues, it is likely that even more efficient and effective algorithms will be developed.

8. REFERENCES

1. Kleinberg, J., & Tardos, E. (2006). *Algorithm design*. Pearson Education India.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.
3. Google
4. Introduction to algorithms Book by Udi Manber

THANK YOU