Bangladesh University Of Engineering & Technology

CSE 300

Technical Writing & Presentation

Report On-

Travelling Salesman Problem(TSP)

Contributed By-

M.M. Nayem-2005078
Kazi Jayed Haider-2005081
Iffat Bin Hossain-2005087

Contents

1 Introduction **4**

2 Problem Formulation **4**

3 Difference between TSP & Hamiltonian Cycle Problem **5**

4 Different Solution Approaches **6**
- 4.1 Exact Algorithms 6
- 4.2 Heuristic Algorithms 6
- 4.3 Approximation Algorithms 6

5 Why TSP cannot be solved with Greedy Algorithm **7**

6 Solving TSP using Branch & Bound **7**
- 6.1 Algorithm 7
- 6.2 Code Implementation 8
- 6.3 Simulation 12
- 6.4 Time Complexity 14

7 Solving TSP using Dynamic Programming(using Bitmask) **14**
- 7.1 Algorithm & pseudocode 14
- 7.2 Code Implementation 15
- 7.3 Simulation 17
- 7.4 Time Complexity 18
- 7.5 Space Complexity 18

8 Solving TSP using Approximation Algorithm **19**
- 8.1 Algorithm 19
- 8.2 Pseudocode 19
- 8.3 Code Implementation 19
  - 8.3.1 Constructing MST 19
  - 8.3.2 Find the PreOrder/DFS walk 20
  - 8.3.3 Final Code 22
- 8.4 Simulation 24
- 8.5 Time Complexity 26
- 8.6 Space Complexity 26

9 Comparison between exact and approximate solution **26**

10 TSP Variants **27**

11 Applications In Practical field **27**

12 Current Trends and Future Directions **28**

13 Conclusion **28**

List of Figures

1 TSP 4
2 TSP VS HCP 5
3 A Graph 12
9 Recursive Definition 14
10 DP Algorithm for solving TSP 15
11 Sample graph 15
12 A random Graph 17
13 Building Recursion Tree 17
14 Building Recursion Tree 17
15 Building Recursion Tree 18
16 Building Recursion Tree 18
17 A graph 25
18 MST 25
19 DFS Traversal 25
20 Optimal Tour 26

Abstract

The Travelling Salesman Problem (TSP) is a classic combinatorial optimization problem with significant real-world applications. This report provides a comprehensive overview of the TSP, discussing its formulation, solution methodologies, variants, applications, and current research trends.

1 Introduction

The Travelling Salesman Problem (TSP) is one of the most studied problems in combinatorial optimization and operations research. It originated in the 19th century and has since garnered immense interest due to its practical relevance in various domains, including logistics, transportation, manufacturing, and more.[12]

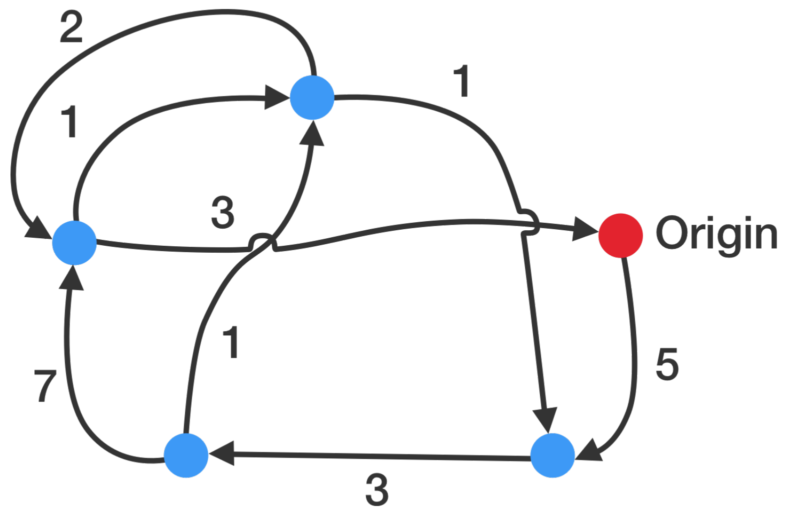

Figure 1: TSP

2 Problem Formulation

The TSP can be formally defined as follows:

Given a set of n cities $C = \{c_1, c_2, ..., c_n\}$ and a distance matrix D, where D_{ij} represents the distance between city c_i and city c_j, the objective is to find the shortest possible tour that visits each city exactly once and returns to the starting city.

Mathematically, the TSP can be formulated as a combinatorial optimization problem:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} D_{ij} x_{ij} \tag{1}$$

subject to:

$$\sum_{i=1}^{n} x_{ij} = 1, \quad \forall j \in \{1, 2, ..., n\} \tag{2}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \quad \forall i \in \{1, 2, ..., n\} \tag{3}$$

$$u_i - u_j + n x_{ij} \leq n - 1, \quad \forall i, j : i \neq 1, j \neq 1, i, j \in \{2, 3, ..., n\} \tag{4}$$

$$2 \leq u_i \leq n, \quad \forall i \in \{2, 3, ..., n\} \tag{5}$$

$$x_{ij} \in \{0, 1\}, \quad \forall i, j \in \{1, 2, ..., n\} \tag{6}$$

Here, x_{ij} is a binary decision variable indicating whether the tour includes the edge from city i to city j, and u_i is a visiting order variable used to ensure that the tour forms a simple cycle.

3 Difference between TSP & Hamiltonian Cycle Problem

The Hamiltonian Cycle Problem (HCP) and Travelling Salesman Problem (TSP) are long-standing and well-known NP-hard problems. The HCP is concerned with finding paths through a given graph such that those paths visit each node exactly once after the start, and end where they began (i.e., Hamiltonian cycles). The TSP builds on the HCP and is concerned with computing the lowest cost Hamiltonian cycle on a weighted (di)graph. Many solutions to these problems exist, including some from the perspective of P systems.[10]

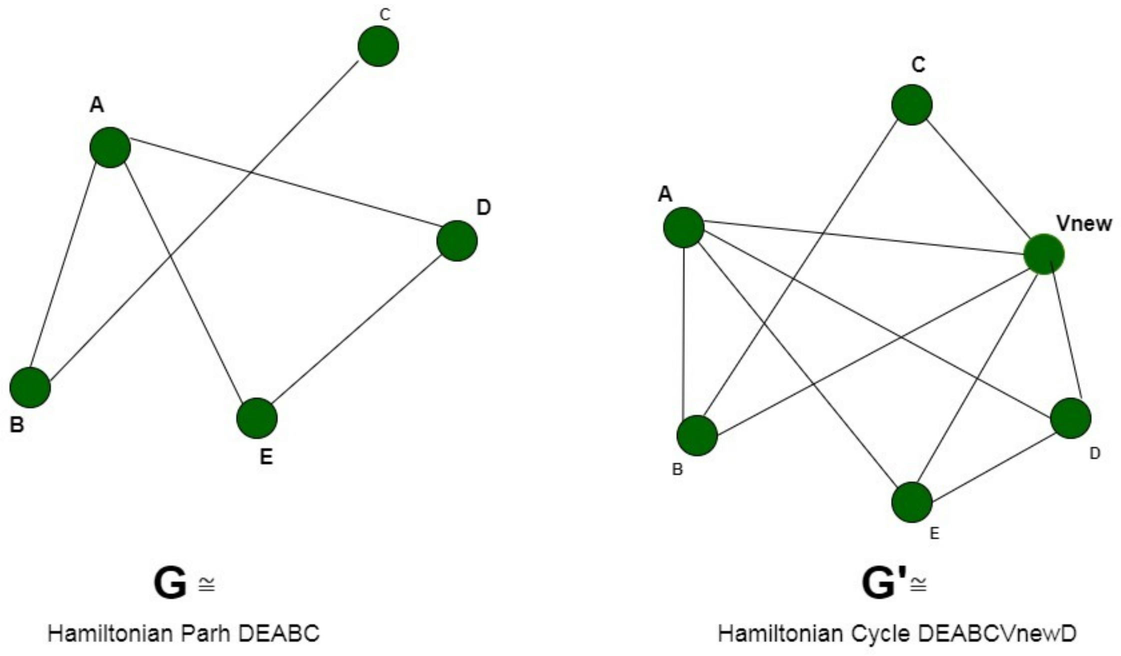

Figure 2: TSP VS HCP

4 Different Solution Approaches

Various solution approaches have been proposed to tackle the TSP, catering to different problem sizes and computational resources. These approaches can broadly be categorized as follows:

4.1 Exact Algorithms

Exact algorithms aim to find the optimal solution to the TSP by exhaustively exploring all possible solutions. Popular exact algorithms include:

- **Brute Force Method**: It involves trying all possible permutations of paths and selecting the shortest one. However, the running time of this method is very large,which means that it quickly becomes impractical even for small instances of the problem.
- **Dynamic Programming(using bitmask)**: Dynamic programming is another technique that can be used to solve the TSP. One of the earliest dynamic programming algorithms for the TSP is the Held-Karp algorithm.However, improving the time complexity beyond this point appears to be quite challenging.
- **Branch and Bound**: A systematic search algorithm that prunes branches of the search tree based on lower bounds on the tour length.

4.2 Heuristic Algorithms

Heuristic algorithms are used to solve NP problems and decrease the time complexity of problems by giving quick solutions.

- **Lin-Kernighan Heuristic**: A local search algorithm that iteratively improves an initial tour to achieve near-optimal solutions.
- **K-opt Heuristics**: Variants of local search algorithms that explore different combinations of edge exchanges to improve tour quality.
- **Match Twice and Stitch (MTS) Heuristics**: The MTS heuristic consists of two phases. In the first phase, known as cycle construction, two sequential matchings are performed to construct cycles. The first matching returns the minimum-cost edge set with each point incident to exactly one matching edge. Next, all these edges are removed, and the second matching is executed. The second matching repeats the matching process, subject to the constraint that none of the edges chosen in the first matching can be used again. Together, the results of the first and second matchings form a set of cycles. The constructed cycles are stitched together in the second phase to form the TSP tour.

4.3 Approximation Algorithms

Approximation algorithms provide approximate solutions to the TSP in a reasonable amount of time. They include:

- **Nearest Neighbor**: The NN algorithm is a greedy approach where the salesman chooses the closest unvisited city as the next destination.
- **Christofides–Serdyukov algorithm**: This algorithm utilizes both the minimum spanning tree and a solution to the minimum-weight perfect matching problem. By combining these two solutions, the algorithm is able to produce a TSP tour that is at most 1.5 times the length of the optimal tour.[1]

5 Why TSP cannot be solved with Greedy Algorithm

In the TSP, a salesman must visit each of a given set of cities exactly once and return to the starting city, while minimizing the total distance traveled.Greedy algorithms are simple, intuitive approaches that make a series of locally optimal choices with the hope that these choices will lead to a globally optimal solution. However, in the case of the TSP, greedy algorithms typically do not guarantee the optimal solution for several reasons:

- **Local Optima**: Greedy algorithms make decisions based on immediate benefits without considering the long-term consequences. This can lead to situations where the algorithm gets stuck in a local optimum, meaning that the solution it finds is not necessarily the best one overall.
- **No Backtracking**: Greedy algorithms do not backtrack or reconsider previous decisions. Once a decision is made, it is generally irreversible. This can cause the algorithm to overlook better solutions that may have been possible by revisiting earlier decisions.
- **Dependence on Starting Point**: The solution obtained by a greedy algorithm for the TSP can depend heavily on the starting point chosen. Since the TSP involves finding the shortest route that visits all cities exactly once, the choice of starting city can significantly affect the overall solution.
- **Not Globally Optimal**: Greedy algorithms may find solutions that are suboptimal or even far from optimal, especially for complex problems like the TSP. While they can work well for certain types of problems, they are not guaranteed to find the best solution for the TSP in all cases.

Due to these limitations, more sophisticated techniques such as dynamic programming, branch and bound, genetic algorithms, or simulated annealing are often used to tackle the TSP and find near-optimal solutions. These approaches can explore the search space more effectively and are better suited to handle the complexities of the TSP.[9]

6 Solving TSP using Branch & Bound

6.1 Algorithm

1. Set an initial value for the best tour cost
2. Initialize the priority queue (PQ)
3. Generate the first node with partial tour [1] and compute its lower bound.
4. Insert this node into the PQ
5. while not empty (PQ)
6. remove the first element in the PQ and assign it to parent node
7. if the lower bound $<$ best tour cost
8. set the level of the node to level of parent node $+1$
9. if this level equals $n-1$ (n being the number of cities)
10. add 1 to the end of the path and compute the cost of the full tour
11. if this cost $<$ best tour cost
12. set the best tour cost and the best tour accordingly

13. else (the level is not equal to $n-1$)

14. for all i such that $2 \leq i \leq n$ and i is not in the path of parent

15. copy the path from parent to the new node

16. add i to the end of this path

17. compute the lower bound for this new node

18. if this lower bound is less than the best tour cost

19. insert this new node into the priority queue

20. end

[2]

6.2 Code Implementation

```
    // 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];

// visited[] keeps track of the already visited nodes
// in a particular path
bool visited[N];

// Stores the final minimum weight of shortest tour.
int final_res = INT_MAX;

// Function to copy temporary solution to
// 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;
}

// function to find the second minimum edge cost
// 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;
}

// 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[]
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
  // 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
      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];

  // 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;
  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));

  // 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
```

```c
  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;
}
```

[5]

6.3 Simulation

Here is the Simulation of B&B algorithm for solving TSP:

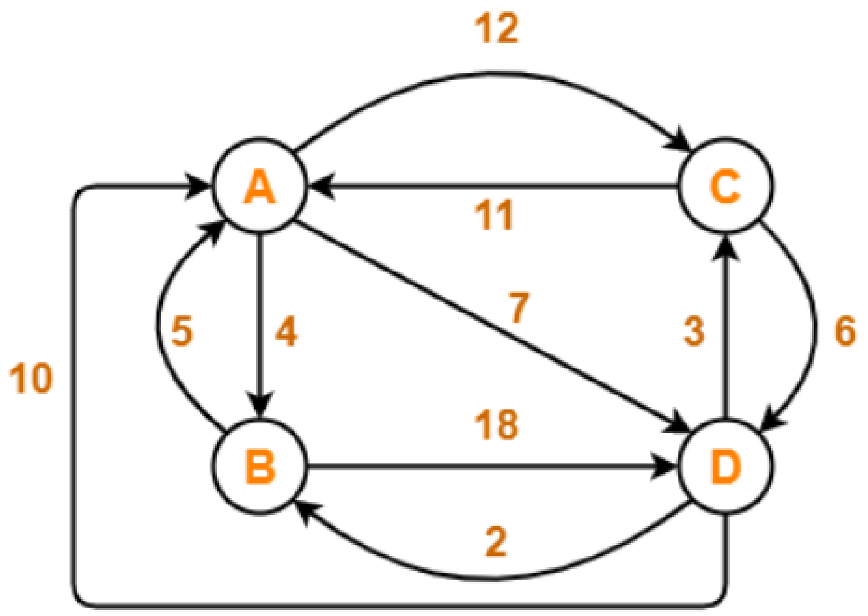

Figure 3: A Graph

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 4 & 12 & 7 \\ B & 5 & \infty & \infty & 18 \\ C & 11 & \infty & \infty & 6 \\ D & 10 & 2 & 3 & \infty \end{array}$$

(a) Initial Cost Matrix

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 0 & 8 & 3 \\ B & 0 & \infty & \infty & 13 \\ C & 5 & \infty & \infty & 0 \\ D & 8 & 0 & 1 & \infty \end{array}$$

(b) Row Reduced Matrix

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 0 & 7 & 3 \\ B & 0 & \infty & \infty & 13 \\ C & 5 & \infty & \infty & 0 \\ D & 8 & 0 & 0 & \infty \end{array}$$

(c) Column Reduced Matrix

[3]

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & 13 \\ C & 5 & \infty & \infty & 0 \\ D & 8 & \infty & 0 & \infty \end{array}$$

(a) Choosing To Go To Vertex-B: Node-2 (Path A $\rightarrow$ B)

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & 0 \\ C & 5 & \infty & \infty & 0 \\ D & 8 & \infty & 0 & \infty \end{array}$$

(b) Row Reduced Matrix

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & 0 \\ C & 0 & \infty & \infty & 0 \\ D & 3 & \infty & 0 & \infty \end{array}$$

(c) Column Reduced Matrix

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & 0 & \infty & \infty & 13 \\ C & \infty & \infty & \infty & 0 \\ D & 8 & 0 & \infty & \infty \end{array}$$

(a) Choosing To Go To Vertex-C: Node-3 (Path A $\rightarrow$ C)

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & 0 & \infty & \infty & \infty \\ C & 5 & \infty & \infty & \infty \\ D & \infty & 0 & 0 & \infty \end{array}$$

(b) Choosing To Go To Vertex-D: Node-4 (Path A $\rightarrow$ D)

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & 0 & \infty & \infty & \infty \\ C & 0 & \infty & \infty & \infty \\ D & \infty & 0 & 0 & \infty \end{array}$$

(c) Row Reduced Matrix

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & 0 & \infty & \infty & 13 \\ C & \infty & \infty & \infty & 0 \\ D & 8 & 0 & \infty & \infty \end{array}$$

Cost(3) = 25

(a) Starting Cost Matrix From Node-3

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & 13 \\ C & \infty & \infty & \infty & \infty \\ D & 8 & \infty & \infty & \infty \end{array}$$

(b) Choosing To Go To Vertex-B: Node-5 (Path A $\rightarrow$ C $\rightarrow$ B)

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & 0 \\ C & \infty & \infty & \infty & \infty \\ D & 0 & \infty & \infty & \infty \end{array}$$

(c) Row Reduced Matrix

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & 0 & \infty & \infty & \infty \\ C & \infty & \infty & \infty & \infty \\ D & \infty & 0 & \infty & \infty \end{array}$$

(a) Choosing To Go To Vertex-D: Node-6 (Path A $\rightarrow$ C $\rightarrow$ D)

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & 0 & \infty & \infty & \infty \\ C & \infty & \infty & \infty & \infty \\ D & \infty & 0 & \infty & \infty \end{array}$$

Cost(6) = 25

(b) Starting Cost Matrix From Node-6

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & \infty \\ C & \infty & \infty & \infty & \infty \\ D & \infty & \infty & \infty & \infty \end{array}$$

(c) Choosing To Go To Vertex-B: Node-7 (Path A $\rightarrow$ C $\rightarrow$ D $\rightarrow$ B)

6.4 Time Complexity

We are actually creating all the possible extensions of E-nodes in terms of tree nodes. Which is nothing but a permutation. Suppose we have N cities, then we need to generate all the permutations of the $(N-1)$ cities, excluding the root city. Hence the time complexity for generating the permutation is $O((n-1)!)$, which is equal to $O(2^{(n-1)})$.

Hence the final time complexity of the algorithm can be $O(n^2 \times 2^n)$.[5]

7 Solving TSP using Dynamic Programming(using Bitmask)

In Bitmask dp method the recursive function will take two parameters. One is current state and another is mask. Mask is a n-bit binary number. Every bit of mask determines whether a city is visited or not. If i-th bit of mask is set that indicates that i-th city is already visited. If i-th bit is reset that means the city is still undiscovered. Inside the recursive function we will visit one of the undiscoverd city, then set that corresponding bit of mask and again call the recursive function with the new mask and new current state. And We will visit the city that will lead us to optimal cost tour. And the base case of the recursion will be when all the bits of mask are set that means $2^n - 1$. Then we will just come back to the starting city and return the cost.

$$f(i, 2^{n-1}) = dis[i][0]$$
$$f(i, mask) = min(f(j, turnOn(mask, j) + w(i, j))\ where(i, j) \in E$$

Figure 9: Recursive Definition

7.1 Algorithm & pseudocode

At first,Let's see the algorithm of solving TSP with DP:
In this algorithm, we take a subset N of the required cities needs to be visited, distance among the cities dist and starting city s as inputs. Each city is identified by unique city id like $\{1, 2, 3, \cdots, n\}$.

Initially, all cities are unvisited, and the visit starts from the city s. We assume that the initial travelling cost is equal to 0. Next, the TSP distance value is calculated based on a recursive function. If the number of cities in the subset is two, then the recursive function returns their distance as a base case.

On the other hand, if the number of cities is greater than 2, then we'll calculate the distance from the current city to the nearest city, and the minimum distance among the remaining cities is calculated recursively.

Finally, the algorithm returns the minimum distance as a TSP solution.

```
Algorithm 1: Dynamic Approach for TSP
  Data: s: starting point; N: a subset of input cities; dist():
        distance among the cities
  Result: Cost : TSP result
  Visited[N] = 0;
  Cost = 0;
  Procedure TSP(N, s)
      Visited[s] = 1;
      if |N| = 2 and k ≠ s then
          Cost(N, k) = dist(s, k);
          Return Cost;
      else
          for j ∈ N do
              for i ∈ N and visited[i] = 0 do
                  if j ≠ i and j ≠ s then
                      Cost(N, j) = min ( TSP(N − {i}, j) + dist(j, i))
                      Visited[j] = 1;
                  end
              end
          end
      end
      Return Cost;
  end
```

Figure 10: DP Algorithm for solving TSP

7.2 Code Implementation

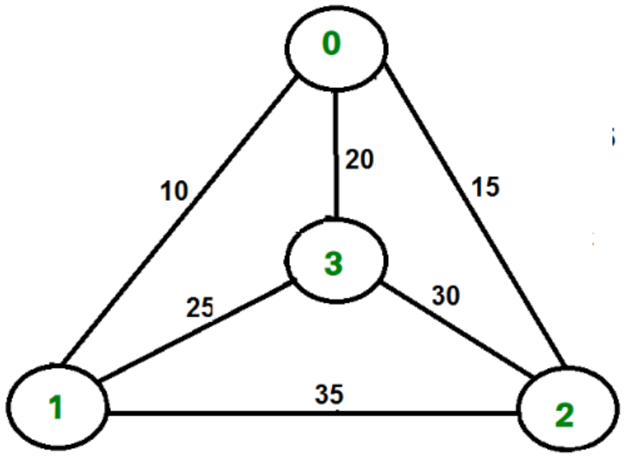

Figure 11: Sample graph

```cpp
#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

  // we have to travel all nodes j in mask and end the
  // path at ith node so for every node j in mask,
  // recursively calculate cost of travelling all nodes in
  // mask except i and then travel back from node j to
  // node i taking the shortest path take the minimum of
  // all possible j nodes

  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;
}
// Driver program to test above logic
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;
}
```

The cost of most efficient tour = 80 for the graph attached above.

7.3 Simulation

Here is the simulation of dynamic approach for solving TSP.

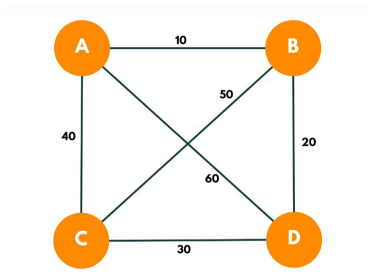

Figure 12: A random Graph

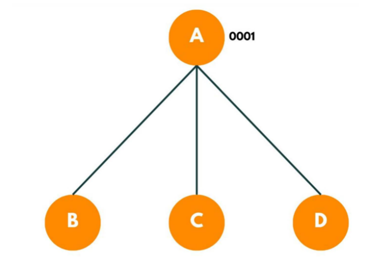

Figure 13: Building Recursion Tree

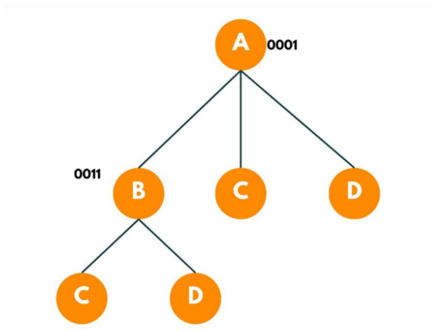

Figure 14: Building Recursion Tree

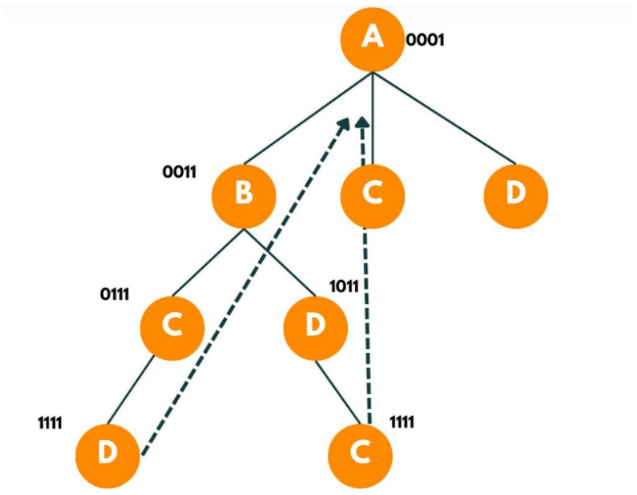

Figure 15: Building Recursion Tree

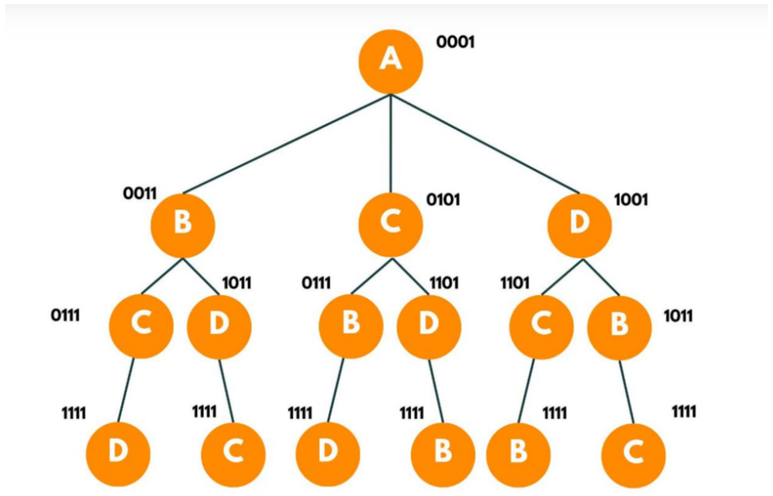

Figure 16: Building Recursion Tree

The shortest distance to visit all the cities is 50.[6]

7.4 Time Complexity

In the dynamic algorithm for TSP, the number of possible subsets can be at most $N \times 2^N$. Each subset can be solved in $\mathcal{O}(N)$ times. Therefore, the time complexity of this algorithm would be $\mathcal{O}(N^2 \times 2^N)$.[6]

7.5 Space Complexity

The time complexity of the dynamic programming algorithm for the Traveling Salesman Problem is $O(n \times 2^n)$, where n is the number of nodes or cities.[6]

8 Solving TSP using Approximation Algorithm

An approximate algorithm is devised only if the cost function (which is defined as the distance between two plotted points) in the problem satisfies the triangle inequality.

The triangle inequality is satisfied if the cost function c for all the vertices of a triangle u, v, and w satisfies the following equation:

$$c(u, w) \leq c(u, v) + c(v, w)$$

[4]

It is usually automatically satisfied in many applications.

8.1 Algorithm

- **Step 1** – Choose any vertex of the given graph randomly as the starting and ending point.
- **Step 2** – Construct a minimum spanning tree of the graph with the vertex chosen as the root using Prim's algorithm.
- **Step 3** – Once the spanning tree is constructed, perform a pre-order traversal on the minimum spanning tree obtained in the previous step.
- **Step 4** – The pre-order solution obtained is the Hamiltonian path of the traveling salesperson.[4]

8.2 Pseudocode

```
 APPROX_TSP(G, c)
r <- root node of the minimum spanning tree
T <- MST_Prim(G, c, r)
visited = {  }
for i in range V:
   H <- Preorder_Traversal(G)
   visited = {H}
```

[11]

1. The cost of the minimum spanning tree (MST) is never less than the cost of the optimal Hamiltonian path: $c(M) \leq c(H^*)$.
2. The cost of a full walk, defined as the path traced while traversing the MST preorderly, is exactly twice the cost of the MST: $c(W) = 2c(T)$, where T is the MST.
3. Since the preorder walk path is less than the full walk path, the output of the algorithm, being a preorder walk, is always lower than the cost of the full walk.

8.3 Code Implementation

We design the code in 3 Steps

8.3.1 Constructing MST

```

int minimum_key(int key[], bool mstSet[])
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
```

```
            min = key[v], min_index = v;

    return min_index;
}

vector<vector<int>> MST(int parent[], int graph[V][V])
{
    vector<vector<int>> v;
    for (int i = 1; i < V; i++)
    {
        vector<int> p;
        p.push_back(parent[i]);
        p.push_back(i);
        v.push_back(p);
        p.clear();
    }
    return v;
}

// getting the Minimum Spanning Tree from the given graph
// using Prim's Algorithm
vector<vector<int>> primMST(int graph[V][V])
{
    int parent[V];
    int key[V];

    // to keep track of vertices already in MST
    bool mstSet[V];

    // initializing key value to INFINITE & false for all mstSet
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // picking up the first vertex and assigning it to 0
    key[0] = 0;
    parent[0] = -1;

    // The Loop
    for (int count = 0; count < V - 1; count++)
    {
        // checking and updating values wrt minimum key
        int u = minimum_key(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }
    vector<vector<int>> v;
    v = MST(parent, graph);
    return v;
}
```

[8]

8.3.2 Find the PreOrder/DFS walk

```

// getting the preorder walk of the MST using DFS
void DFS(int** edges_list,int num_nodes,int starting_vertex,bool* visited_nodes)
{
    // adding the node to final answer
    final_ans.push_back(starting_vertex);

```

```cpp
    // checking the visited status
    visited_nodes[starting_vertex] = true;

    // using a recursive call
    for(int i=0;i<num_nodes;i++)
    {
        if(i==starting_vertex)
        {
            continue;
        }
        if(edges_list[starting_vertex][i]==1)
        {
            if(visited_nodes[i])
            {
                continue;
            }
            DFS(edges_list,num_nodes,i,visited_nodes);
        }
    }
}
int main()
{
    // initial graph
    int graph[V][V] = { { 0, 10, 18, 40, 20 },
                        { 10, 0, 35, 15, 12 },
                        { 18, 35, 0, 25, 25 },
                        { 40, 15, 25, 0, 30 },
                        { 20, 13, 25, 30, 0 } };

    vector<vector<int>> v;

    // getting the output as MST
    v = primMST(graph);

    // creating a dynamic matrix
    int** edges_list = new int*[V];
    for(int i=0;i<V;i++)
    {
        edges_list[i] = new int[V];
        for(int j=0;j<V;j++)
        {
            edges_list[i][j] = 0;
        }
    }

    // setting up MST as adjacency matrix
    for(int i=0;i<v.size();i++)
    {
        int first_node = v[i][0];
        int second_node = v[i][1];
        edges_list[first_node][second_node] = 1;
        edges_list[second_node][first_node] = 1;
    }

    // a checker function for the DFS
    bool* visited_nodes = new bool[V];
    for(int i=0;i<V;i++)
    {
        bool visited_node;
        visited_nodes[i] = false;
    }

    //performing DFS
    DFS(edges_list,V,0,visited_nodes);

    // adding the source node to the path
    final_ans.push_back(final_ans[0]);

    // printing the path
    for(int i=0;i<final_ans.size();i++)
    {
        cout << final_ans[i] << "-";
    }
    return 0;
}
```

[8]

8.3.3 Final Code

```cpp

#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// Dynamic array to store the final answer
vector<int> final_ans;

int minimum_key(int key[], bool mstSet[])
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

vector<vector<int>> MST(int parent[], int graph[V][V])
{
    vector<vector<int>> v;
    for (int i = 1; i < V; i++)
    {
        vector<int> p;
        p.push_back(parent[i]);
        p.push_back(i);
        v.push_back(p);
        p.clear();
    }
    return v;
}

// getting the Minimum Spanning Tree from the given graph
// using Prim's Algorithm
vector<vector<int>> primMST(int graph[V][V])
{
    int parent[V];
    int key[V];

    // to keep track of vertices already in MST
    bool mstSet[V];

    // initializing key value to INFINITE & false for all mstSet
    for (int i = 0; i < V; i++)
```

```
        key[i] = INT_MAX, mstSet[i] = false;

    // picking up the first vertex and assigning it to 0
    key[0] = 0;
    parent[0] = -1;

    // The Loop
    for (int count = 0; count < V - 1; count++)
    {
        // checking and updating values wrt minimum key
        int u = minimum_key(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }
    vector<vector<int>> v;
    v = MST(parent, graph);
    return v;
}

// getting the preorder walk of the MST using DFS
void DFS(int** edges_list,int num_nodes,int starting_vertex,bool* visited_nodes)
{
    // adding the node to final answer
    final_ans.push_back(starting_vertex);

    // checking the visited status
    visited_nodes[starting_vertex] = true;

    // using a recursive call
    for(int i=0;i<num_nodes;i++)
    {
        if(i==starting_vertex)
        {
            continue;
        }
        if(edges_list[starting_vertex][i]==1)
        {
            if(visited_nodes[i])
            {
                continue;
            }
            DFS(edges_list,num_nodes,i,visited_nodes);
        }
    }
}
int main()
{
    // initial graph
    int graph[V][V] = { { 0, 10, 18, 40, 20 },
                        { 10, 0, 35, 15, 12 },
                        { 18, 35, 0, 25, 25 },
                        { 40, 15, 25, 0, 30 },
                        { 20, 13, 25, 30, 0 } };

    vector<vector<int>> v;

    // getting the output as MST
    v = primMST(graph);

    // creating a dynamic matrix
    int** edges_list = new int*[V];
    for(int i=0;i<V;i++)
    {
        edges_list[i] = new int[V];
        for(int j=0;j<V;j++)
        {
            edges_list[i][j] = 0;
        }
    }

    // setting up MST as adjacency matrix
    for(int i=0;i<v.size();i++)
    {
        int first_node = v[i][0];
        int second_node = v[i][1];
        edges_list[first_node][second_node] = 1;
        edges_list[second_node][first_node] = 1;
    }

    // a checker function for the DFS
    bool* visited_nodes = new bool[V];
    for(int i=0;i<V;i++)
    {
        bool visited_node;
        visited_nodes[i] = false;
    }

    //performing DFS
    DFS(edges_list,V,0,visited_nodes);

    // adding the source node to the path
    final_ans.push_back(final_ans[0]);

    // printing the path
    for(int i=0;i<final_ans.size();i++)
    {
        cout << final_ans[i] << "-";
    }
    return 0;
}
```

[8]

8.4 Simulation

1. Consider a graph.
2. Construct the Minimum Spanning Tree (MST) from the graph.
3. Perform Depth-First Search (DFS) Traversal.
4. Get the Optimal Tour.

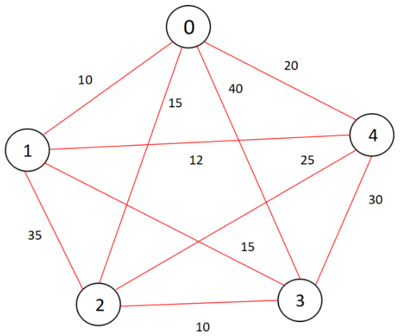

Figure 17: A graph

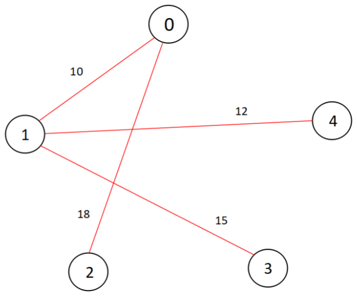

Figure 18: MST

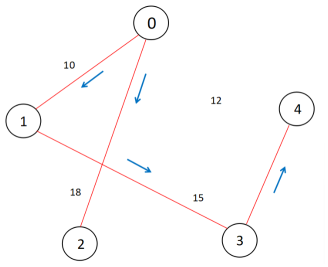

Figure 19: DFS Traversal

Hence we have the optimal path according to the approximation algorithm, i.e., $0 \to 1 \to 3 \to 4 \to 2 \to 0$.

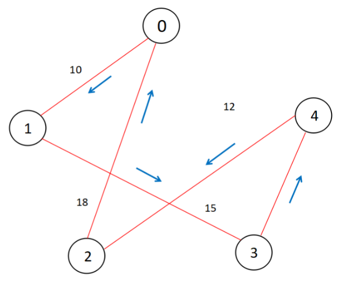

Figure 20: Optimal Tour

8.5 Time Complexity

[4]

The time complexity for obtaining the Minimum Spanning Tree (MST) from the given graph using algorithms like Prim's or Kruskal's is indeed $O(V^2)$, where V is the number of nodes. However, this complexity can be reduced to $O(V \log V + E)$ using more efficient algorithms like Prim's or Kruskal's with data structures like priority queues or disjoint-set data structures. For Depth First Search (DFS) of a graph, the time complexity is $O(V + E)$, where V is the number of nodes and E is the number of edges. This is because in the worst case, every node and every edge may need to be explored.

8.6 Space Complexity

As for the space complexity, constructing a vector of vectors (`vector<vector<int>>`) to store the final MST would indeed require $O(V^2)$ space, as you're essentially representing the MST as an adjacency matrix.

9 Comparison between exact and approximate solution

- $Cost_{\text{Best possible Travelling Salesman tour}} \geq Cost_{\text{MST}}$.
- The definition of MST says, it is a minimum cost tree that connects all vertices.
- $Cost_{\text{Best possible Travelling Salesman tour}} \leq 2 \cdot Cost_{\text{MST}}$.
- Every edge of MST is visited at most twice.

10 TSP Variants

Several variants and extensions of the classical TSP exist, each with its own unique characteristics and solution challenges. Some notable variants include:

- **Multiple TSP**: Involves visiting multiple sets of cities, each with its own salesman, while minimizing the total distance traveled.
- **Time-Dependent TSP**: Considers varying travel times between cities due to factors such as traffic conditions or time windows for visiting cities.
- **Asymmetric TSP**: The distance between two cities may differ depending on the direction of travel, leading to asymmetric distance matrices.
- **Vehicle Routing Problem with TSP Constraints**: Extends the TSP to include additional constraints such as vehicle capacities and multiple depots.

Each variant presents unique challenges and requires tailored solution methodologies.

11 Applications In Practical field

The Travelling Salesman Problem finds applications in diverse fields, demonstrating its versatility and importance. Some prominent applications include:

1. **Logistics and Supply Chain Management:** The TSP is often used to optimize delivery routes and minimize travel costs for logistics and supply chain management. For example, companies like UPS and FedEx use TSP algorithms to optimize their delivery routes and schedules.
2. **DNA Sequencing:** The TSP has been used in DNA sequencing to determine the order in which to fragment a DNA molecule to obtain its complete sequence. TSP algorithms can help minimize the number of fragments required to sequence a DNA molecule.
3. **Circuit Board Design:** TSP algorithms can be used to optimize the design of circuit boards, where the goal is to minimize the length of the connections between the components.
4. **Network Design:** The TSP can be used to design optimal networks for telecommunications, transportation, and other systems. It can help optimize the placement of facilities and routing of traffic in such systems.
5. **Manufacturing and Production Planning:** The TSP can be used in production planning and scheduling to optimize the order in which different tasks are performed, minimize setup times, and reduce production costs.
6. **Robotics:** The TSP can be used to optimize the path of a robot as it moves through a series of tasks or objectives. TSP algorithms can help minimize the distance traveled by the robot and reduce the time required to complete the tasks.
7. **Image and Video Processing:** The TSP can be used to optimize the order in which different parts of an image or video are processed. This can help improve the efficiency of image and video processing algorithms.

These applications highlight the practical significance of the TSP in addressing complex optimization challenges across various domains.[7]

12 Current Trends and Future Directions

Recent advancements in algorithms and computational techniques have enabled the solution of larger and more complex instances of the TSP. Key trends and future directions in TSP research include:

- **Hybrid Algorithms**: Combining different solution methodologies to exploit their complementary strengths and improve solution quality.
- **Parallel and Distributed Computing**: Leveraging parallel computing architectures to accelerate solution times for large-scale TSP instances.
- **Integration with Real-Time Data**: Incorporating real-time data on traffic conditions, customer demands, and other dynamic factors to enhance solution accuracy and adaptability.
- **Multi-Objective Optimization**: Considering multiple conflicting objectives such as cost, time, and environmental impact in TSP solutions to achieve more sustainable outcomes.

These trends indicate a continued focus on developing innovative approaches to tackle the TSP and its variants in increasingly challenging and dynamic environments.[7]

13 Conclusion

The Travelling Salesman Problem is a fundamental problem in combinatorial optimization with wide-ranging applications across various industries. Despite its computational complexity, significant progress has been made in developing efficient solution methodologies to address both classical and variant instances of the TSP. As research in this field continues to evolve, the TSP remains a fertile ground for exploration, innovation, and practical problem-solving.

References

[1] Baeldung. Tsp: Exact solutions vs. heuristic vs. approximation algorithms, 2024.

[2] James Cooper and Radu Nicolescu. Bio-inspired computing: Theories and applications (bic-ta 2017). *Journal Name*, Volume Number(Issue Number):Page Range, 2017. Guest editors: Linqiang Pan, Mario J. Pérez-Jiménez and Gexiang Zhang.

[3] GateVidyalay. Travelling salesman problem using branch and bound approach, 2024.

[4] GeeksforGeeks. Approximate solution for travelling salesman problem using mst, 2024.

[5] GeeksforGeeks. Traveling salesman problem using branch and bound, 2024.

[6] GeeksforGeeks. Travelling salesman problem using dynamic programming, 2024.

[7] OpenAI. Chatgpt. OpenAI's Language Model, 2022.

[8] OpenGenus. Approximation algorithm for travelling salesman problem, 2024.

[9] Stack Overflow. Travelling salesperson: Why greedy algos are not guaranteed to give an optimal, 2024.

[10] Robert Sedgewick. Algorithms in c++: The traveling salesperson problem. *Journal of Object Technology*, 2(3):127–137, 2003.

[11] Tutorialspoint. Travelling salesman approximation algorithm, 2024.

[12] Wikipedia contributors. Travelling salesman problem, 2024.