A

MINI PROJECT REPORT ON

"Different exact and approximation algorithms for Travelling-Sales-Person Problem"

Submitted to the Department of Computer Engineering,

SMT.KASHIBAI NAVALE COLLEGE OF ENGINEERING,PUNE

LABORATORY PRACTICE - III

Design & Analysis of Algorithm

FINAL YEAR (COMPUTER ENGINEERING)

By

Rushikesh Hole **C41125**

Yash Jadhav **C41128**

Aniket Kadlag **C41132**

Under the guidance of

Prof.Priyanka Kinage

DEPARTMENT OF COMPUTER ENGINEERING

SMT.KASHIBAI NAVALE COLLEGE OF ENGINEERING, PUNE

2024 – 2025

Sinhgad Institutes

SAVITRIBAI PHULE PUNE UNIVERSITY. 2024-25

CERTIFICATE

This is to certify that the Internship report entitles

"Different exact and approximation algorithms for Travelling-Sales-Person Problem"

Submitted by

Rushikesh Hole	**C41125**
Yash Jadhav	**C41128**
Aniket Kadlag	**C41132**

is a bonafide student of this institute and the work has been carried out by him/her under the supervision of **Prof. Priyanka Kinage.** This work is approved for the partial fulfillment of the requirement of Savitribai Phule Pune University, for the award of the degree of **Bachelor of Engineering** (Computer Engineering)

Prof. Priyanka Kinage
Guide ,
Department of Computer Engineering

Prof. R. H.Borhade
Vice Principal &
Head of Computer Engg. Department

Dr. A. V. Deshpande
Principal,
Smt. Kashibai Navale College of Engineering Pune – 411046.

Place : Pune

Date:

CONTENTS

Abstract

This report provides an overview of exact and approximation algorithms for the Traveling Salesperson Problem (TSP). Exact algorithms explored include Brute Force, Dynamic Programming, and Branch and Bound, offering guaranteed optimal solutions. For approximations, the Nearest Neighbor Algorithm and Minimum Spanning Tree Algorithms are examined, providing faster solutions with trade-offs in optimality. The report aids in understanding algorithmic choices based on problem size and computational needs, offeringinsights for practical TSP problem-solving.

Introduction

The Traveling Salesperson Problem (TSP) represents a quintessential optimization challenge with applications spanning logistics, circuit design, and beyond. It involves determining the shortest path that visits a set of cities exactly once and returns to the starting point. Given the NP-hard nature of TSP, devising efficient algorithms is paramount.

This report provides a focused exploration of both exact and approximation algorithms for TSP. The exact algorithms—Brute Force, Dynamic Programming, and Branch and Bound— offer solutions with optimality guarantees. On the approximation side, the Nearest Neighbor Algorithm and Minimum Spanning Tree Algorithms provide expedient solutions, albeit with potential trade-offs in optimality. This investigation aims to equip practitioners and researchers with insights into the strengths and limitations of these algorithms, facilitating informed decisions in navigating the complexities of TSP problem-solving.

Problem Statement

The Traveling Salesperson Problem (TSP) is a fundamental optimization challenge, tasked with determining the most efficient route for a salesperson to visit a predetermined set of cities precisely once, concluding the journey by returning to the initial point of departure. This problem is formally represented as a complete graph, where cities are denoted as nodes, and edges between them represent the distances to be traversed. The core objective of TSP centers around minimizing the total distance traveled along a Hamiltonian cycle—a cyclic paththat visits each city exactly once

Motivation

The Traveling Salesperson Problem (TSP) is a pervasive challenge with direct applications in logistics, network design, and beyond. Addressing TSP efficiently is crucial for cost reduction and resource optimization. This report is motivated by the practical relevance of TSP, aiming to explore algorithms that not only contribute to theoretical advancements but also offer tangible solutions for real-world optimization problems.

Theory

In the realm of algorithmic solutions for the Traveling Salesperson Problem (TSP), the primary goal is to find the optimal route for visiting a set of cities exactly once and returning to the starting point. This is formalized through the representation of cities as nodes in a complete graph, where edges denote the distances between them. Tackling TSP involves the exploration of exact algorithms such as Brute Force, Dynamic Programming, and Branch and Bound, ensuring optimality. Additionally, approximation algorithms like Nearest Neighbor and Minimum Spanning Tree offer efficient, albeit approximate, solutions. The evaluation of these algorithms encompasses considerations of computational efficiency, optimality, and scalability, providing valuable insights for practical problem-solving scenarios.

Algorithm Selection:

Exact Algorithms: Algorithms such as Brute Force, Dynamic Programming, and Branch and Bound guarantee optimal solutions.

Approximation Algorithms: Heuristic methods like Nearest Neighbor and Minimum Spanning Tree algorithms provide efficient, albeit approximate, solutions.

Exact Algorithms:

1. Brute Force:

Description: The brute force algorithm for the Traveling Salesperson Problem (TSP) exhaustively generates all possible permutations of cities and calculates the total distance for each permutation.

- Pros: Guarantees an optimal solution.
- Cons: Computationally expensive with a time complexity of O(n!).

2. Dynamic Programming (DP):

Description: The dynamic programming algorithm for TSP uses the principle of optimality to avoid redundant calculations by storing and reusing intermediate results.

- Pros: More efficient than brute force for larger instances.
- Cons: Still exponential time complexity, but with a smaller constant factor.

3. Branch and Bound:
Description: Branch and Bound is an exact algorithm that prunes the search space using bounds to eliminate subproblems that cannot lead to an optimal solution.

- Pros: More efficient than brute force for large instances.
- Cons: Time complexity depends on the quality of the bounding function.

Approximation Algorithms:

1. Nearest Neighbor Algorithm:
Description: The Nearest Neighbor Algorithm starts from a randomly chosen city and repeatedly selects the nearest unvisited city until all cities are visited.

- Pros: Simple and fast.
- Cons: Does not always produce optimal solutions.

2. Minimum Spanning Tree (MST) Algorithms:
Description: Minimum Spanning Tree algorithms build a minimum spanning tree of the cities and traverse the tree to form a tour.

- Pros: Efficient and often provides good approximations.
- Cons: Not guaranteed to be optimal.

Result:-

Code-

Brute Force Algorithm:

```
from itertools import permutations

def total_distance(path, distances):
    return sum(distances[path[i]][path[i + 1]] for i in range(len(path) - 1))

def brute_force_tsp(distances):
    n = len(distances)
    all_paths = permutations(range(n))
    min_path = min(all_paths, key=lambda path: total_distance(path, distances))
    return min_path, total_distance(min_path, distances)

# Example Usage:
distances = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

min_path, min_distance = brute_force_tsp(distances)
print("Optimal Path:", min_path)
print("Optimal Distance:", min_distance)
```

Dynamic Programming Algorithm:

```
def tsp_dynamic_programming(distances):
    n = len(distances)
    memo = {}

    def dp(mask, pos):
        if mask == (1 << n) - 1:
            return distances[pos][0]
```

```
        ans = float('inf')
            for next_pos in range(n):
                if mask & (1 << next_pos) == 0:
                    ans = min(ans, distances[pos][next_pos] + dp(mask | (1 << next_pos),
        next_pos))

            memo[(mask, pos)] = ans
            return ans

        return dp(1, 0)

    # Example Usage:
    min_distance_dp = tsp_dynamic_programming(distances)
    print("Optimal Distance (Dynamic Programming):", min_distance_dp)
```

Branch and Bound Algorithm:

```
import heapq

def tsp_branch_and_bound(distances):
    n = len(distances)
    pq = [(0, 0, 1)] # (cost, current_vertex, visited_mask)
    best_cost = float('inf')

    while pq:
        cost, current_vertex, visited_mask = heapq.heappop(pq)

        if visited_mask == (1 << n) - 1:
            best_cost = min(best_cost, cost)
            continue

        for next_vertex in range(n):
            if not (visited_mask & (1 << next_vertex)):
                new_cost = cost + distances[current_vertex][next_vertex]
                heapq.heappush(pq, (new_cost, next_vertex, visited_mask | (1 <<
next_vertex)))

    return best_cost

# Example Usage:
min_distance_bb = tsp_branch_and_bound(distances)
print("Optimal Distance (Branch and Bound):", min_distance_bb)
```

Nearest Neighbor Algorithm:

```
def nearest_neighbor_tsp(distances):
    n = len(distances)
    unvisited = set(range(1, n))
    path = [0]

    while unvisited:
        current_city = path[-1]
        nearest_city = min(unvisited, key=lambda city:
distances[current_city][city])
        path.append(nearest_city)
        unvisited.remove(nearest_city)

    path.append(0) # Return to the starting city
    total_distance = sum(distances[path[i]][path[i + 1]] for i in range(n))

    return path, total_distance

# Example Usage:
path_nn, distance_nn = nearest_neighbor_tsp(distances)
print("Approximate Path (Nearest Neighbor):", path_nn)
print("Approximate Distance (Nearest Neighbor):", distance_nn)
```

Minimum Spanning Tree (MST) Algorithm:

```
import networkx as nx

def mst_tsp(distances):
    n = len(distances)
    graph = nx.Graph()

    for i in range(n):
        for j in range(i + 1, n):
            graph.add_edge(i, j, weight=distances[i][j])
```

```
    mst_edges = nx.minimum_spanning_edges(graph, algorithm='kruskal', data=False)
    mst_path = list(mst_edges)
    mst_path.append(mst_path[0]) # Complete the cycle
    total_distance = sum(distances[i][j] for i, j in zip(mst_path, mst_path[1:]))
    return mst_path, total_distance
# Example Usage:
path_mst, distance_mst = mst_tsp(distances)
print("Approximate Path (MST):", path_mst)
print("Approximate Distance (MST):", distance_mst)
```

Brute Force Algorithm:

```python
def total_distance(path, distances):
    return sum(distances[path[i]][path[i + 1]] for i in range(len(path) - 1))

def brute_force_tsp(distances):
    n = len(distances)
    all_paths = permutations(range(n))
    min_path = min(all_paths, key=lambda path: total_distance(path, distances))
    return min_path, total_distance(min_path, distances)

# Example Usage:
distances = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

min_path, min_distance = brute_force_tsp(distances)
print("Optimal Path:", min_path)
print("Optimal Distance:", min_distance)
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
PS C:\Users\ayush\OneDrive\Desktop\qr code generaator> python -u "c:\Users\ayush\OneDrive\Desktop\qr code generaator\aaa.py"
Optimal Path: (2, 0, 1, 3)
Optimal Distance: 50
PS C:\Users\ayush\OneDrive\Desktop\qr code generaator>
```

Dynamic Programming Algorithm:

```python
def tsp_dynamic_programming(distances):
    n = len(distances)
    memo = {}

    def dp(mask, pos):
        if mask == (1 << n) - 1:
            return distances[pos][0]
        if (mask, pos) in memo:
            return memo[(mask, pos)]
        ans = float('inf')
        for next_pos in range(n):
            if mask & (1 << next_pos) == 0:
                ans = min(ans, distances[pos][next_pos] + dp(mask | (1 << next_pos), next_pos))
        memo[(mask, pos)] = ans
        return ans
    return dp(1, 0)

distances = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]]
min_distance_dp = tsp_dynamic_programming(distances)
print("Optimal Distance (Dynamic Programming):", min_distance_dp)
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\ayush\OneDrive\Desktop\qr code generaator> python -u "c:\Users\ayush\OneDrive\Desktop\qr code generaator\aaa.py"
Optimal Distance (Dynamic Programming): 80
PS C:\Users\ayush\OneDrive\Desktop\qr code generaator>
```

Branch and Bound Algorithm:

qr code generaator > aaa.py

```python
import heapq

def tsp_branch_and_bound(distances):
    n = len(distances)
    pq = [(0, 0, 1)]  # (cost, current_vertex, visited_mask)
    best_cost = float('inf')

    while pq:
        cost, current_vertex, visited_mask = heapq.heappop(pq)

        if visited_mask == (1 << n) - 1:
            best_cost = min(best_cost, cost)
            continue

        for next_vertex in range(n):
            if not (visited_mask & (1 << next_vertex)):
                new_cost = cost + distances[current_vertex][next_vertex]
                heapq.heappush(pq, (new_cost, next_vertex, visited_mask | (1 << next_vertex)))

    return best_cost
distances_bb = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

min_distance_bb = tsp_branch_and_bound(distances_bb)
print("Optimal Distance (Branch and Bound):", min_distance_bb)
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
Optimal Distance (Branch and Bound): 65
PS C:\Users\ayush\OneDrive\Desktop\qr code generaator>
```

Nearest Neighbor Algorithm:

```python
def nearest_neighbor_tsp(distances):
    n = len(distances)
    unvisited = set(range(1, n))
    path = [0]

    while unvisited:
        current_city = path[-1]
        nearest_city = min(unvisited, key=lambda city: distances[current_city][city])
        path.append(nearest_city)
        unvisited.remove(nearest_city)

    path.append(0)  # Return to the starting city
    total_distance = sum(distances[path[i]][path[i + 1]] for i in range(n))

    return path, total_distance

# Example Usage:
distances_nn = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

path_nn, distance_nn = nearest_neighbor_tsp(distances_nn)
print("Approximate Path (Nearest Neighbor):", path_nn)
print("Approximate Distance (Nearest Neighbor):", distance_nn)
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\ayush\OneDrive\Desktop\qr code generaator> python -u "c:\Users\ayush\OneDrive\Desktop\qr code generaator\aaa.py"
Approximate Path (Nearest Neighbor): [0, 1, 3, 2, 0]
Approximate Distance (Nearest Neighbor): 80
```

Minimum Spanning Tree(MST) Algorithm:

```
    graph = nx.Graph()
    for i in range(n):
        for j in range(i + 1, n):
            graph.add_edge(i, j, weight=distances[i][j])

    mst_edges = nx.minimum_spanning_edges(graph, algorithm='kruskal', data=False)
    mst_path = list(mst_edges)
    mst_path.append(mst_path[0])  # Complete the cycle

    # Corrected code: Use i, j = mst_path[index] to get indices
    total_distance = sum(distances[i][j] for i, j in mst_path)

    return mst_path, total_distance

# Example Usage:
distances_mst = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

path_mst, distance_mst = mst_tsp(distances_mst)
print("Approximate Path (MST):", path_mst)
print("Approximate Distance (MST):", distance_mst)
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\ayush\OneDrive\Desktop\qr code generaator>
                                                    > python -u "c:\Users\ayush\OneDrive\Desktop\qr code generaator\aaa.py"
Approximate Path (MST): [(0, 1), (0, 2), (0, 3), (0, 1)]
Approximate Distance (MST): 55
```

Conclusion

These algorithms offer a spectrum of approaches to solve the Traveling Salesperson Problem, each with its trade-offs in terms of computational complexity and solution quality. The choice between exact and approximation algorithms depends on the specific requirements of the problem, considering factors such as the size of the input, the need for an optimal solution, and the available computational resources. As TSP is an NP-hard problem, exact algorithms are essential for small instances where optimality is critical, while approximation algorithms provide scalable solutions for larger instances.

References

- TA. G. Little, K. E. Murty, D. W. Sweeney, and C. Karel, "An Algorithm for the Traveling Salesman Problem," Operations Research, 1963.

- M. L. Fisher, D. L. Schrager, and J. K. MacCarthy, "The Structure of the Set Covering Problem with Applications to the Design of Packing Algorithms," Operations Research, 1981.

- H. W. Hamacher, "A Note on the Nearest Neighbor Algorithm forthe Traveling Salesman Problem," Operations Research Letters, 1985.

- R. L. Graham, "An Efficient Algorithm for Determining the ConvexHull of a Finite Planar Set," Information Processing Letters, 1972.

- C. H. Papadimitriou and K. Steiglitz, "Combinatorial Optimization: Algorithms and Complexity," Prentice-Hall, 1982.