Comparing Algorithms for TSP Optimization

Rhea Ajaykumar Patel
Computer Engineering
NMIMS Navi Mumbai
Navi Mumbai, India
rheapatel2607@gmail.com

Anindya Sanjeev Zarbade
Computer Engineering
NMIMS Navi Mumbai
Navi Mumbai, India
anindyasanjeev24@gmail.com

Variza Negi
Assistant Professor, Mentor
NMIMS Navi Mumbai
Navi Mumbai, India
variza.negi@nmims.edu

***Abstract*—There exists a set of instructions to obtain the most suitable outcome for a given problem. These set of instructions are called algorithms. Several types of algorithms implement various approaches to solve a problem ranging from brute force algorithms that iterate every possibility available to recursive algorithms. It is important to analyze the efficiency of the algorithm as it dictates how quickly the problem will be solved and how many resources the algorithm will utilize to achieve the desired outcome. Our paper focuses on the research and analysis of Recursive Problems mainly greedy and dynamic algorithms where it compares their space and time complexity to determine which algorithm has the least resource utilization and time usage. Analysis has been done using the same dataset on multiple algorithms using the R and Python programming languages.**

***Index Terms*—Time Complexity, Space Complexity, Dynamic Algorithm, Greedy Algorithm, Algorithm Analysis**

I. INTRODUCTION

An algorithm is a finite set of instructions where each instruction has a clear meaning and can be performed in finite amount of effort taking finite amount of time. For example, assignment of an integer statement like a = b+c is an instruction that can be executed in finite amount of time and is understandable by a human with finite amount of effort. Instructions in an algorithm can be executed any number of times. However, it is important that no matter what the input values are, the algorithm shall terminate in a finite amount time.

[1] Thus a program is an algorithm until it enters an infinite loop on any input.

Distance-finding algorithms are the base of many modern technologies like GPS navigation systems, robotics, and geographic information systems. They are used to determine the distance between two points in space, which is essential for accurate positioning and navigation. We can use various algorithms to find the distance between two or many points like the A* algorithm, Prim's Algorithm, Flyod-Warshall Algorithm, a very well-known Dijkstra's Algorithm, and many more.[2] Distance finding algorithms are important because they allow us to find the shortest path between some or various points in a graph which is subsequently useful in many real-world applications. For example, in transportation, (Fig 1) distance-finding algorithms can be used to find the shortest route between two locations. In networking, they can be used to find the shortest path between nodes (Fig 2) in anetwork, and in robotics, for the robot to find the shortest path to do work or reach the destination.

Fig. 1. Image Showing Shortest Path Between Two Locations

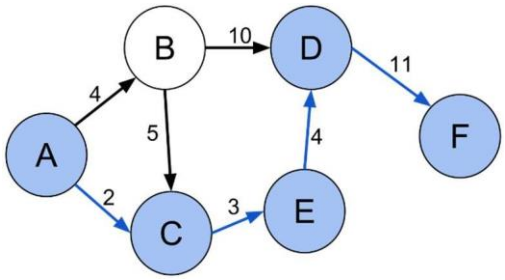

Fig. 2. Image Showing Shortest Path Between Two Nodes

Distance-finding algorithms play a crucial role in various fields, ranging from logistics and transportation to network architecture and analysis and computational biology. In transportation networks, distance-finding algorithms help to optimize the route for vehicles to reach their destinations efficiently, thus reducing travel time and fuel consumption. They aid in finding the shortest path between nodes, which is crucial for network performance and minimizing data transmission delays.

In the logistics and Supply chain management sector these algorithms help in optimizing the flow of goods and services, thereby reducing operational cost, and improving customer satisfaction. Also used in warehouse management to get the shortest path for picking and packing items, helping reduce the time taken for order fulfillment. In geographic information systems (GIS), distance-finding algorithms are used to compute.

979-8-3503-8719-3/24/$31.00 ©2024

distances between geographic locations, which is essential for accurate mapping and spatial analysis. They also provide real-time directions to users.

In computational biology, they are used to analyze biological data, such as DNA and protein sequences, to identify evolutionary relationships and similarities between distinct species also play a crucial role in phylogenetic analysis, where they help in constructing evolutionary trees to understand the genetic history of organisms. These algorithms complement pathfinding and navigation for creating virtual environments and making them more efficient. In game AI it is used to simulate real-world behaviors, such as finding the minimum path to a target or avoiding obstacles.

A. *Optimization*

Finding the minimum and shortest distance between points or nodes is crucial for optimizing various processes, such as transportation routes network paths, and resource allocation.

B. *Efficiency*

By minimizing distances, one can achieve greater efficiency in terms of time, cost, and resource optimization, leading to improved performance and productivity.

C. *Accuracy*

Identifying the shortest paths and minimum distances allows for better allocation of resources, reducing wastage, and ensuring optimal use of available resources which will consecutively help in making informed decisions and developing effective strategies.

II. TYPES OF ALGORITHMS

A. *Divide and Conquer*

This variety of algorithms works by breaking down the problem into smaller and more manageable sub-problems/parts where each sub-problem is solved independently, and then the solutions are merged to obtain one final solution. It divides an array into smaller arrays based on a pivot element and then recursively sorts the two smaller arrays.

B. *Greedy Algorithm*

These algorithms work by making a series of decisions that, at the time of making each decision, seem to be the best possible choice. They do not reconsider earlier choices once they are made. While these algorithms are simple and easy to implement, they often do not guarantee an optimal solution.[4] A generalized example is the minimum spanning tree (MST) problem, where the greedy strategy is to select the cheapest edge that repeatedly connects two disjointed trees until all the vertices are connected.

C. *Dynamic Algorithm*

This approach is based on solving problems by breaking them down into simpler sub-problems and storing their solutions so that they can be reused later. Dynamic programming is beneficial when the same sub-problem occurs multiple times. One example is the Fibonacci sequence, where dynamic programming can be used to avoid redundant calculations of the same Fibonacci numbers.

D. *Backtracking*

This variety of algorithms systematically searches for all possible solutions to a problem. They do this by exploring all paths, and if a path is found to be incorrect or leads to a dead end, the algorithm backtracks and tires a different path. For example, the N-Queen problem, where the algorithm tries to arrange Y queens on a YxY chessboard so that two queens attack each other.

III. DATASET

In this paper, we will be focusing on the greedy and dynamic algorithms where we will be comparing the types of these algorithms concerning their time and space complexities using two datasets named [5] *att48_xy*, a 48-city dataset having X and Y coordinates of places and *p01_sxy_15*, a 15-city dataset again having X and Y coordinates. Here is a small representation of our data in the form of the 15-city dataset in the form of (Fig 3) table and (Fig 4) map. The map was made using a Python code in Google Collab [6] generated as an HTML output.

-0.0000000400893815	0.0000000358808126
-21.4983260706612533	-7.3194159498090388
-28.8732862244731230	-0.0000008724121069
-43.0700258454450875	-14.5548396888330487
-50.4808382862985496	-7.3744722432402208
-64.7472605264735108	-21.8981713360336698
-72.0785319657452987	-0.1815834632498404
-79.2915791686897506	21.4033307581457670
-65.0865638413727368	36.0624693073746769
-57.5687244704708050	43.2505562436354225
-50.5859026832315024	21.5881966132975371
-36.0366489745023770	21.6135482886620949
-29.0584693142401171	43.2167287683090606
-14.6577381710829471	43.3895496964974043
-0.1358203773809326	28.7292896751977480
-0.0000000400893815	0.0000000358808126

Fig. 3. Image Showing Dataset as Co-Ordinates

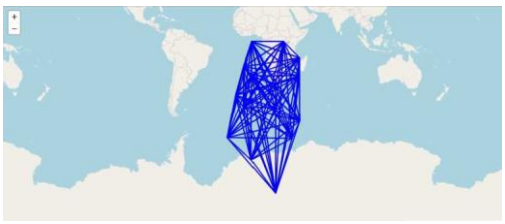

Fig. 4. Image Showing Map Representation of 15 City Dataset

IV. COMPARISON AND CONTRAST

In this section, we will be implementing the 4 types of greedy algorithms and 2 types of dynamic algorithms using the 2 datasets mentioned in Section 3 of our paper.

Since both the datasets have X and Y coordinates of the places, we will be using Euclidian's Formula to calculate the distances between two points and form a distance matrix for the same which will be further used in the implementation of different algorithms to compare the time and space complexities of those algorithms.

The Euclidean method is the most straightforward way to calculate the distance between two points in a two-dimensional plane (such as a map).[7] The formula to calculate the Euclidean distance between two points assuming the points are (x1, y1) and (x2, y2) is:

$$\sqrt{(x1 - x2) + (y1 - y2)}$$

We have implemented the codes in two different PCs for comparative goals. Now, Let's see the outputs and analyze them.

A. 15 City Dataset

1) Dijkstra's Algorithm: Implementing Dijkstra's Algorithm using the Euclidean method on two different PCs, Table I and Table II show you the results for the same.

TABLE I
DIJKSTRA'S ALGORITHM ON 15 CITY USING EUCLIDEAN METHOD

Rhea's Windows 10 Intel i5-6200U CPU @ 2.40 GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Dijkstra's	54.81	2264

TABLE II

Anindya's Windows 10 Intel i5-7200U CPU @ 2.50GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Dijkstra's	44.27	2264

Conclusion: *Anindya's CPU is faster by 10.77% (9.54 ms less) in terms of time.*

2) Prim's Algorithm: Implementing Prim's Algorithm using the Euclidean method on two different PCs, Table III and Table IV show you the results for the same.

TABLE III
PRIM'S ALGORITHM ON 15 CITY USING EUCLIDEAN METHOD

Rhea's Windows 10 Intel i5-6200U CPU @ 2.40 GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Prim's	42.17	2264

TABLE IV

Anindya's Windows 10 Intel i5-7200U CPU @ 2.50GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Prim's	32.25	2264

Conclusion:
Anindya's CPU is faster by 23.94% (9.92 ms less) in terms of time.

3) Kruskal's Algorithm: Implementing Kruskal's Algorithm using the Euclidean method on two different PCs, Table V and Table VI show you the results for the same.

TABLE V
KRUSKAL'S ALGORITHM ON 15 CITY USING EUCLIDEAN METHOD

Rhea's Windows 10 Intel i5-6200U CPU @ 2.40 GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Kruskal's	30.30	872

TABLE VI

Anindya's Windows 10 Intel i5-7200U CPU @ 2.50GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Kruskal's	36.1	872

Conclusion:
Rhea's CPU is faster by 15.25% (5.8 ms less) in terms of time.

4) Knapsack's Algorithm: Implementing Knapsack's Algorithm using the Euclidean method on two different PCs, Table VII, and Table VIII show you the results for the same.

TABLE VII
KNAPSACK'S ALGORITHM ON 15 CITY USING EUCLIDEAN METHOD

Rhea's Windows 10 Intel i5-6200U CPU @ 2.40 GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Knapsack's	17.33	523

TABLE VIII

Anindya's Windows 10 Intel i5-7200U CPU @ 2.50GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Knapsack's	17.8	523

Conclusion:
Rhea's CPU is faster by 2.4% (0.47112 ms less) in terms of time.

5) Bellman-Ford Algorithm: Implementing Bellman-Ford Algorithm using the Euclidean method on two different PCs, Table IX and Table X show you the results for the same.

TABLE IX
BELLMAN-FORD ALGORITHM ON 15 CITY USING EUCLIDEAN METHOD

Rhea's Windows 10 Intel i5-6200U CPU @ 2.40 GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Bellman-Ford	35.25	176

TABLE X

Anindya's Windows 10 Intel i5-7200U CPU @ 2.50GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Bellman-Ford	44.27	2264

Conclusion: *Rhea's CPU is faster by 8.42% (3.03292 ms less) in terms of time.*

6) Floyd's Algorithm: Implementing Floyd's Algorithm using the Euclidean method on two different PCs, table XI and Table XII show you the results for the same.

TABLE XI
FLOYD'S ALGORITHM ON 15 CITY USING EUCLIDEAN METHOD

Rhea's Windows 10 Intel i5-6200U CPU @ 2.40 GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Floyd's	23.30	2264

TABLE XII

Anindya's Windows 10 Intel i5-7200U CPU @ 2.50GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Floyd's	25.95	2264

Conclusion:
Rhea's CPU is faster by 10.53% (2.65694 ms less) in terms of time.

B. 48 City Dataset

1) Dijkstra's Algorithm: Implementing Dijkstra's Algorithm using the Euclidean method on two different PCs, table XIII and Table XIV show you the results for the same.

TABLE XIII
DIJKSTRA'S ALGORITHM ON 48 CITY USING EUCLIDEAN METHOD

Rhea's Windows 10 Intel i5-6200U CPU @ 2.40 GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Dijkstra's	52.49	18648

TABLE XIV

Anindya's Windows 10 Intel i5-7200U CPU @ 2.50GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Dijkstra's	68.33	18648

Conclusion: *Anindya's CPU is slower by 23.836 ms (45.6% slower) in terms of time.*

2) Prim's Algorithm: Prim's Algorithm: Implementing Prim's Algorithm using the Euclidean method on two different PCs, Table XV and Table XVI show you the results for the same.

TABLE XV
PRIM'S ALGORITHM ON 48 CITY USING EUCLIDEAN METHOD

Rhea's Windows 10 Intel i5-6200U CPU @ 2.40 GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Prim's	55.03	18648

TABLE XVI

Anindya's Windows 10 Intel i5-7200U CPU @ 2.50GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Prim's	55.60	18648

Conclusion: *Anindya's CPU is slower by 0.56489 ms (1.03% slower) in terms of time.*

3) Kruskal's Algorithm: Kruskal's Algorithm: Implementing Kruskal's Algorithm using the Euclidean method on two different PCs, Table XVII and Table XVIII show you the results for the same.

TABLE XVII
KRUSKAL'S ALGORITHM ON 48 CITY USING EUCLIDEAN METHOD

Rhea's Windows 10 Intel i5-6200U CPU @ 2.40 GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Kruskal's	45.57	1000

TABLE XVIII

Anindya's Windows 10 Intel i5-7200U CPU @ 2.50GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Kruskal's	48.31	1000

Conclusion: *Rhea's CPU is faster by 2.74515 ms (6.02% faster) in terms of time.*

4) Knapsack's Algorithm: Knapsack's Algorithm: Implementing Knapsack's Algorithm using the Euclidean method on two different PCs, Table XIX and Table XX show you the results for the same.

TABLE XIX
KNAPSACK'S ALGORITHM ON 48 CITY USING EUCLIDEAN METHOD

Rhea's Windows 10 Intel i5-6200U CPU @ 2.40 GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Knapsack's	17.15	6464

TABLE XX

Anindya's Windows 10 Intel i5-7200U CPU @ 2.50GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Knapsack's	18.57	6464

Conclusion:
Anindya's CPU is slower by 1.41501 ms (8.24% slower) interms of time.

5) Bellman-Ford Algorithm: Bellman-Ford Algorithm: Implemented Bellman-Ford Algorithm using the Euclidean method on two different PCs, Table XXI and Table XXII show you the results for the same.

TABLE XXI
BELLMAN-FORD ALGORITHM ON 48 CITY USING EUCLIDEAN METHOD

Rhea's Windows 10 Intel i5-6200U CPU @ 2.40 GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Bellman-Ford	54.64	432

TABLE XXII

Anindya's Windows 10 Intel i5-7200U CPU @ 2.50GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Bellman-Ford	44.27	432

Conclusion: *Rhea's CPU is faster by 20.40195 ms (37.64% faster) in terms of time.*

6) Floyd's Algorithm: Floyd's Algorithm: Implementing Floyd's Algorithm using the Euclidean method on two different PCs, Table XXIII and Table XXIV show you the results for the same.

TABLE XXIII
FLOYD'S ALGORITHM ON 48 CITY USING EUCLIDEAN METHOD

Rhea's Windows 10 Intel i5-6200U CPU @ 2.40 GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Floyd's	162.73	18648

TABLE XXIV

Anindya's Windows 10 Intel i5-7200U CPU @ 2.50GHz		
Euclidean	***Time(ms)***	***Space(bytes)***
Floyd's	182.64	18648

Conclusion:
Rhea's CPU is faster by 19.9139 ms (38.07% faster)in terms of time.

V. ANALYSIS OF THE ALGORITHMS

Now, let us analyze the algorithms based on our outputs using graphs to conclude.

A. For Execution Time

Using the output data found in (4), we plotted a bar graph from which conclusions can be drawn.

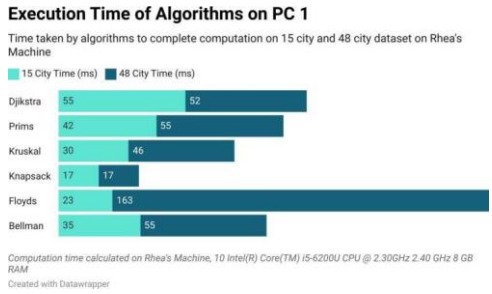

Fig. 5. Time Complexity on PC 1

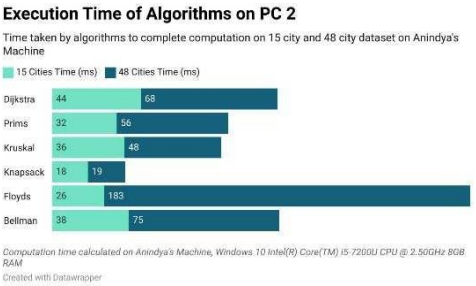

Fig. 6. Time Complexity on PC 2

Based on the two graphs, we can conclude that for the given data sets and based on the average times, we can rank the algorithms in terms of their time complexities:

1) Knapsack's Algorithm (Lowest Average Time)
2) Kruskal's Algorithm
3) Dijkstra's Algorithm
4) Bellman Ford's Algorithm
5) Prim's Algorithm
6) Floyd's Algorithm (Highest Average Time)

Note: *These conclusions are only for the TSP problems and may differ as the problem statement and input change. The specific characteristics of the datasets and PCs used for computation can also influence the results.*

Now, based on the two PCs, calculate the total execution time by each algorithm for Anindya's PC:

- Dijkstra's Algorithm: 44+ 68 = 112 ms
- Prim's Algorithm: 32+ 56 = 88 ms
- Kruskal's Algorithm: 36+48 = 84 ms
- Knapsack's Algorithm: 18+19 = 37 ms
- Floyd's Algorithm: 26+183 = 209 ms
- Bellman Ford's Algorithm: 38+75 = 113 ms

Calculate the execution time of each algorithm for Rhea's PC:

- Dijkstra's Algorithm: 55+ 53 = 108 ms
- Prim's Algorithm: 42+ 55 = 97 ms
- Kruskal's Algorithm: 30+46 = 76 ms
- Knapsack's Algorithm: 17+18 = 35 ms
- Floyd's Algorithm: 23+163 = 186 ms
- Bellman Ford's Algorithm: 35+55 = 90 ms

From the above calculation, we can observe that as the test bench changes the outputs vary which indicates that the outputs of the algorithms may vary concerning several factors.

like hardware, cache behavior, operating system, compiler, and optimization, etc.

B. For Space Complexity

We can conclude that for the given data sets, we can rank the algorithms in terms of their space complexities:

- Bellman: 176 bytes
- Knapsack: 523 bytes
- Kruskal: 872 bytes
- Prims: 2264 bytes
- Floyd's: 2264 bytes
- Dijkstra: 2264 bytes

C. For Time Complexity

Evaluating the codes for each algorithm and analyzing the Big Oh (Worst case) time complexity graph,

- Bellman Ford: $O(n^3)$
  - Creating a matrix: $O(n^2)$
  - Algorithm 3 nested loops: $O(n^3)$
  - Checking for negative cycles: $O(n^2)$
- Knapsack: $O(n \log n)$
  - Calculating the ratio of value to weight: $O(n)$
  - Sorting items based on ratios: $O(n \log n)$
  - Main loop: $O(n)$
  - where n = number of items
- Kruskal: $O(m \log m)$
  - Sorting m edges: $O(m \log m)$
  - Finding parent of the vertex: $O(1)$
  - Loop over all the edges: $O(m)$
  - Overall: $O(n^2 \log m)$
  - where (n^2) directly proportional to (m) and n = number of vertices
- Prims: $O(n^2)$
  - 2 matrix loops: $O(n^2)$
- Floyds: $O(n^3)$
  - Creating a matrix: $O(n^2)$
  - 3 nested loops: $O(n^3)$
- Dijkstra: $O(n^2)$
  - Creating a matrix: $O(n^2)$
  - 2 nested loops: $O(n^2)$

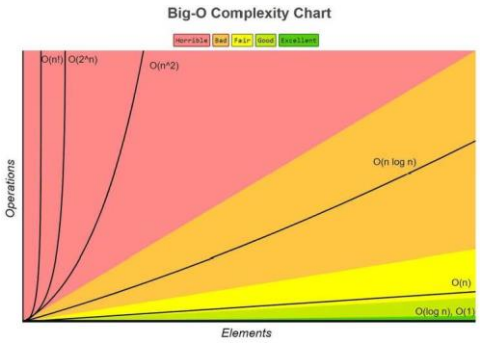

Fig. 7. Big Oh Complexity Chart

We can conclude that for the given data sets, we can rank the algorithms in terms of their time complexities:

- Kruskal: $O(m \log m)$
- Knapsack: $O(n \log n)$
- Prims: $O(n^2)$
- Dijkstra: $O(n^2)$
- Bellman-ford: $O(n^3)$
- Floyd: $O(n^3)$

Note: These conclusions are only for the TSP problems and may differ as the problem statement and input change. The specific characteristics of the datasets and PCs used for computation can also influence the results. The codes for these implementations are available on the GitHub repository.

VI. CONCLUSION

Based on the data derived, concerning execution times, space complexities, and time complexities for each algorithm on both Anindya's and Rhea's PCs, we can draw the following conclusions

- Dijkstra's Algorithm:
  - It consistently performs well in terms of time efficiency on both PCs, taking 112 ms on Anindya's PC and 96 ms on Rhea's PC.
  - However, its space complexity is higher compared to some other algorithms.
  - It has a quadratic time complexity which is most suitable for dense graphs like in GPS navigation systems.

- Prim's Algorithm:
  - Shows superior performance in terms of time efficiency, taking 88 ms on both PCs.
  - Its space complexity is lower compared to Dijkstra's but higher than Knapsack, Kruskal, and Bellman-Ford.
  - Quadratic time complexity is suitable to be used in dense graphs, like used in data transmission and traffic flow.

- Kruskal's Algorithm:
  - Kruskal's Algorithm demonstrates efficient time performance, with 84 ms on Anindya's PC and 82 ms on Rhea's PC.
  - It has a lower space complexity compared to Prim's and Dijkstra's.
  - Sorting edges dominate time complexity and can be used in sparse graphs to create the minimum spanning tree such as in telecommunication networks.

- Knapsack Algorithm:
  - This algorithm shows the best execution efficiency among all considered algorithms, taking 37 ms on Anindya's PC and 35 ms on Rhea's PC.
  - It also has the lowest space complexity among the algorithms considered.
  - Most suitable for optimization problems or resource allocation problems like project scheduling, budget planning, etc.

- Floyd's Algorithm:
  - Floyd's Algorithm exhibits slower performance compared to other algorithms, with 209 ms on Anindya's PC and 189 ms on Rhea's PC.
  - Its space complexity is higher.

- Less efficient for larger graphs, can be used in All-pair shortest path problems like network optimization, traffic flow analysis, etc.

- Bellman Ford's Algorithm:
  - Bellman Ford's Algorithm's performance is decent in terms of time efficiency, with 113 ms on Anindya's PC and 93 ms on Rhea's PC.
  - Its space complexity is also relatively low.
  - Cubic Time complexity which makes it less efficient for large graphs but can be used in networking routing where edges represent cost or distance with negative values.

Considering both time and space complexities, the Knapsack Algorithm comes out as the most efficient choice for the given datasets and TSP optimization. It consistently shows the fastest time performance on both PCs and has the lowest space complexity. However, it is important to note that the choice of algorithm depends on the specific problem requirements, input size, and constraints. While Knapsack might be the best for this scenario, other algorithms may be more suitable for different contexts.

LIMITATIONS AND FUTURE SCOPE

The limitation of this paper is that the findings regarding the execution time of those algorithms are easily influenced by the state of the machine. More specifically, how much resource is currently available for the program to consume. Background processes and programs will affect the runtime of those algorithms. The results also vary according to the test bench being used. A better-configured machine will provide better results. The greedy Algorithm provides an efficient approach and solution to the Travelling Salesman Problem (TSP), but it fails to guarantee optimal routes. This inability leaves room for future advancements. One way to improve might be utilizing both Greedy and Dynamic Algorithms in a hybrid manner. Problem-specific approaches to greedy algorithms and new optimization techniques can enhance its efficiency.[9]

Dynamic Algorithms become very computationally expensive as the dimensions and the size of the dataset increase. Future Enhancing more efficient data structure and implementation of machine learning and artificial learning techniques can enhance the adaptability and scalability of algorithmic variations.

As the size and complexity of TSP instances continue to grow, there is a pressing need for algorithms that can efficiently manage massive datasets while maintaining acceptable levels of performance. This makes it necessary to explore advanced data structures, parallel computing techniques, and optimization strategies tailored specifically for large-scale TSP problems.

Exploring novel algorithms in TSP optimization, such as Genetic Algorithms, and Particle Swarm Optimization, like newly explored algorithms presents exciting opportunities for advancing routing efficiency. These innovative approaches offer potential improvements in solving complex TSP instances, opening new opportunities for research and practical applications in various domains.

REFERENCES

1. *Some books are cited as:*

   [1] Aho, Alfred V., et al." Data Structures and Algorithms." Addison- Wesley, 1983.

   [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition. MIT Press, 2009. [A classic textbook on algorithms, providing a comprehensive introduction to various algorithms and their complexities.]

   [3] Introduction to Algorithms (3rd Edition) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

2. *Database source is cited as:*

   [4] Institute für Informatik, TSPLIB95 datasets, 15- city, 48-city :1995.

   [5] 15 city map representation, Google Colab

3. *A Conference Paper is cited as:*

   [6] N. Ekanayake and R. Liyanapathirana," On the exact formula for the minimum squared Euclidean distance of CPFSK," in IEEE Transactions on Communications, vol. 42, no. 11, pp. 2917 2918, Nov. 1994.

   [7] Sarma, A. D., et al." Weighted clustering algorithms for minimum energy consumption in wireless sensor networks." Proceedings of the1st ACM international symposium on mobile ad hoc networking and computing. 2003. 300-312. [A paper on energy-efficient clustering algorithms in wireless sensor networks, published in the proceedings of a top conference (MobiHoc) by the Association for Computing Machinery (ACM).]

4. *A Journal Paper is cited as:*

   [8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

   [9] GitHub Repository Link: https://github.com/anindyaTHEgrt/DAA Research Paper.git