Extended Essay Computer Science

Topic: Dijkstra vs Bellman-Ford Algorithm

Research Question: *"To what extent does Dijkstra's algorithm outperform the Bellman-Ford algorithm in terms of time complexity while solving the Traveling Salesman Problem as a special case of the Complex graph search algorithm?"*

Word count: 3996

Table of Contents

1.Introduction 4

1A. Plan of Investigation 6

2. Graph Search Algorithm 7

2A. Graphs 7

2B.1. Directed graphs 8

2B.2 Weighted graphs 8

2C. Traveling Salesman Problem (TSP) 9

3. Time complexity 11

2A.1 The Big O notation 11

4. Algorithms 14

4A. Bellman-ford algorithm 14

4B. Dijkstra's Algorithm 15

4B.1 Calculating the theoretical time complexity 16

5. Comparison 18

5A. Primary investigation 18

5B. Data collection 19

6. Result Analysis 20

7. Conclusion 22

8. Bibliography 25

8A. Books 25

8B. Websites 25

8C. Images 27

9. Glossary 29

9A. Terminologies 29

10. Appendices 31

10A. Specifications of the computer system 31

10B. Table of Raw Data 31

10.C Image of the generated Graph 43

10C. Source code of the program used 44

Table of Figures

Figure 1: A graph displaying the worst-case time complexity and runtime 12

Figure 2: Example of an graph 7

Figure 3: Example of a directed graph 8

Figure 4: Example of a directed graph 9

Figure 5: Example of an Travelling Salesmen Problem 10

Figure 6: The Pseudocode of Bellman ford algorithm 14

Figure 7: The Pseudocode of Dijkstra's algorithm 16

Figure 8: Time complexity of the algorithm 17

Figure 9: Time complexity of Dijkstra's algorithm 18

Figure 10: Graph generated for the TSP considering all capitals of India 43

Table of Equations

Equation 1: Asymptotic behaviour of a function 11

Equation 2: Big O notation properties 13

Equation 3: Edge density equation 18

Equation 4: 1 - Relation of E and V 19

Equation 5: 2 - Relation of E and V 19

1.Introduction

One of the most important fields of study in computer science is the efficiency of algorithms. Algorithms are the bedrock of problem-solving, they enable programmes to carry out different tasks. There are a set of instructions and rules that must be adhered to when performing calculations or carrying out other problem-solving methods[1]. To find the efficiency of an algorithm the standard method is to calculate the time complexity of the algorithm. An algorithm's time complexity demonstrates how the size of the input influences the amount of computer resources needed for it to be executed. Because of their complex nature, graph search algorithms are very important in the field of algorithms. Furthermore, this complex Graph Search is a key method for solving a wide range of real-world issues and is considered a foundational approach to problem-solving. Graphs involve recognising specific paths or connections between nodes, which are made up of nodes (vertices) connected by links (edges)[2]. Graph search algorithms systematically explore these graphs, computing optimal paths based on predefined criteria. These algorithms hold a critical position in solving intricate computational problems, like determining the shortest path for network routing or solving puzzles. Despite the versatility of graph search algorithms, their application, especially those involving complex recursive functions, might not always provide accurate real-time reflections. One of the most used graph search algorithms is the Travelling Salesman Problem. It finds the shortest possible path that visits a given set of cities and comes back to the origin, covering each city only once. The computational resources increase dramatically as the number of cities grows. This exponential increase in the number of possible paths makes it difficult to

[1] Nikolopoulou, Kassiani. “What Is an Algorithm? | Definition & Examples.” Scribbr, 9 August 2023, https://www.scribbr.com/ai-tools/what-is-an-algorithm/. Accessed 3 April 2023.
[2] Wikipedia, https://adacomputerscience.org/concepts/struct_graph?examBoard=all&stage=all. Accessed 29 April 2023.

solve. The complexity inherent in these algorithms poses challenges for precise runtime estimations[3]. Hence this problem is chosen as a base to be solved using different algorithms

This insight proves vital in understanding an algorithm's capacity to handle larger problem instances. Typically, scenarios are categorized into worst and best cases. The worst-case scenario denotes the upper bound— the maximum time an algorithm consumes for execution given the worst possible input[4]. Calculating this worst-case scenario is critical, assuring that the algorithm won't surpass the calculated upper limit. Conversely, best-case scenarios represent the minimal time an algorithm expends in execution under the most favourable input conditions.

Primary objective centres on comparing the efficiency of Bellman Ford and Dijkstra's Algorithms while scrutinizing how closely the actual runtime aligns with the theoretically calculated time complexity for these NP-hard problems. The intriguing appeal of the Bellman-Ford and Dijkstra algorithms prompts me to unveil the practical realities of their runtimes in real-world scenarios. Contemplating the intricacies of Hamiltonian paths, especially within the context of India's capitals, sheds light on the tangible applicability of these algorithms. This link to real-world situations gives our research significance and closes the gap between theoretical abstraction and useful applications.

Consider the Traveling Salesman Problem (TSP), a classic puzzle where efficiency is not merely a theoretical concept but a pressing real-world need. Unravelling the real-world runtimes of algorithms Bellman-Ford and Dijkstra hold promise in enhancing our understanding of these algorithms, potentially leading to more efficient solutions in the

[3] Md, Sandeeb. ",." , - YouTube, 21 October 2023, https://link.springer.com/article/10.1007/s10703-019-00337-w. Accessed 4 May 2023.

[4] Kharwal, Aman. "Worst Case, Average Case, and Best Case | Aman Kharwal." thecleverprogrammer, 9 November 2020, https://thecleverprogrammer.com/2020/11/09/worst-case-average-case-and-best-case/. Accessed April 17 2023.

complex landscape of computational challenges[5]. The potential applications of these findings resonate across industries, spanning logistics, network design, and resource optimization, among others. Furthermore, dissecting the nexus between time complexity and actual runtime empowers informed algorithm selection for specific tasks, posing the research question:

"To what extent does Dijkstra's algorithm outperform the Bellman-Ford algorithm in terms of time complexity while solving complex problems like the Traveling Salesman Problem as a special case of the Complex Graph Search Algorithm?"

1A. Plan of Investigation

A initial investigation is planned to gather the amount of time that each of the two algorithms used to solve the Travelling Salesman Problem. The following is how the steps are organised:

1. Create a dataset representing the map of India.
2. Define nodes within the dataset, with each node representing a capital city in India.
3. Choose Bellman-Ford and Dijkstra algorithms to solve the Travelling Salesman Problem (TSP) and Hamiltonian path problems (HPP).
4. Utilize NetworkX in Python for graph-related tasks. Generate data structures for graphs, digraphs, and multigraphs.
5. Implement the selected Bellman-Ford and Dijkstra algorithms.
6. Ensure adjustments are made to these algorithms to efficiently handle the created dataset representing India's capital cities.
7. Execute the implemented algorithms on the dataset. Record the execution times.
8. Calculate the expected time complexities for the Bellman-Ford and Dijkstra algorithms.
9. Compare the actual runtime obtained with the theoretically calculated time complexities.

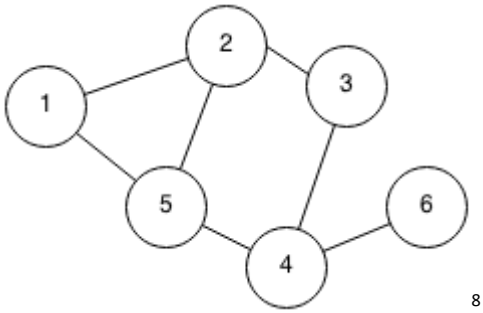
[5] Jain, Sandeep. "What are the differences between Bellman Ford's and Dijkstra's algorithms?" GeeksforGeeks, 23 June 2022, https://www.geeksforgeeks.org/what-are-the-differences-between-bellman-fords-and-dijkstras-algorithms/. Accessed 10 May 2023.

10. Conclude the investigation by summarizing the findings and their implications for solving TSP in the context of India's capital cities.

2. Graph Search Algorithm

2A. Graphs

Graphs are non-linear data structures formed by connected edges and nodes, or vertices[6]. Vertices are uniquely indexed and can have finite or infinite weights. These represent the weights (efforts) required to transverse them. Directed graphs specify the direction of edges. Weighted graphs are edges with associated weight, representing the cost of traversing them[7]. These are applied to real-world problems like the Travelling salesman, where we consider forces like distance, time, and cost.



[8]

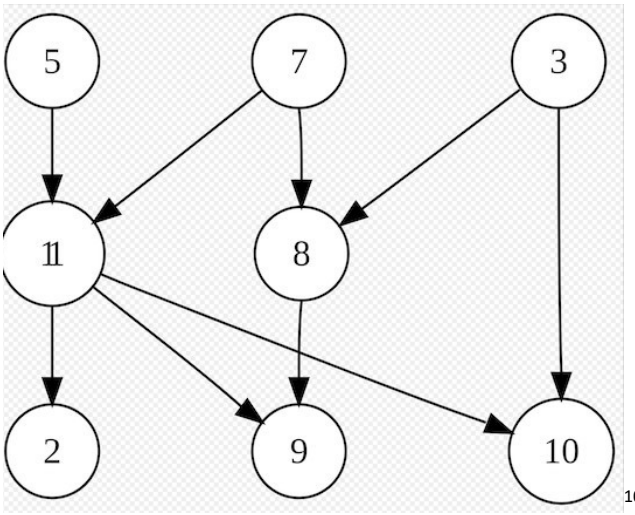
Figure 1: Example of a graph

To calculate the shortest path for example: If a path has the transverse vertices as {v0, v1, v2, ..., vk}, The weight of the path as a whole is equal to the weight of each component (w(p)). The one with the minimum cumulative weight (δ(u, v)) is the shortest path and it represents the optimal route.

[6] "Graphs - Intro to Data Structures." Codology, https://www.codology.org/intro-to-data-structures/graphs. Accessed 18 August 2023.
[7] Md, Sandeeb. ",." , - YouTube, 21 October 2023, https://www.sciencedirect.com/topics/mathematics/weighted-graph. Accessed 21 August 2023.
[8] Young, Michael. "Graph Theory." The Computer Science Handbook, https://www.thecshandbook.com/Graph_Theory. Accessed 21 August 2023.

2B.1. Directed graphs

Graphs can be divide into two categories based on the nature of their edges: directed graphs and undirected graphs. In undirected graphs, movement between vertices is allowed in both directions along an edge. On the other hand, in directed graphs, movement is restricted to the specified direction of the edge[9]. For this experiment, directed graphs are chosen for their utility. This preference arises because directed edges can emulate undirected edges effectively. This is achieved by introducing two oppositely directed edges. This selection allows for a more practical and insightful exploration in the experiment, leveraging the flexibility of directed graphs in simulating undirected scenarios when necessary.



[10]

Figure 2: Example of a directed graph

2B.2 Weighted graphs

By giving weights to edges that indicate the cost of navigating those edges, weighted graphs add a layer of complexity. The traveling salesman problem is one of the real-world situations

[9] "Directed and Undirected Graphs - MATLAB & Simulink." MathWorks, https://www.mathworks.com/help/matlab/math/directed-and-undirected-graphs.html. Accessed 22 August 2023.

[10] Rössel, Johannes. "File:Directed acyclic graph 2.svg." Wikipedia, https://en.m.wikipedia.org/wiki/File:Directed_acyclic_graph_2.svg. Accessed 24 August 2023.

where these weighted edges are useful. The salesman's job in this scenario is to find the best path while taking into account the costs, times, and distances connected to each edge.

A weight function defined as w: E R for a given graph G = (V, E) makes it easier to apply real values to these weights; in this case, V and E represent the graph's vertices and edges, respectively[11]. This weight function serves as a mechanism to quantify and express the real-world considerations involved in traversing the edges of the graph.

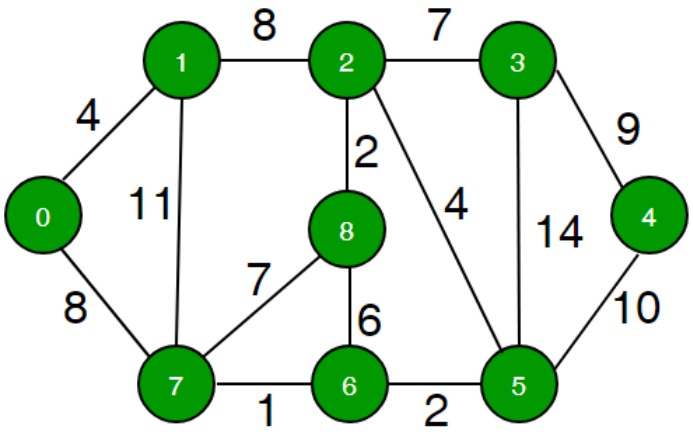


Figure 3: Example of a directed graph[12]

2C. Traveling Salesman Problem (TSP)

The Traveling Salesman Problem (TSP) is all about developing a path through a graph that goes through every vertex. The is the purpose of the traveling salesman problem (TSP) is to find a path that traverses every vertex exactly once. It finds the shortest possible path which visits a given set of cities and comes back to the origin, covering each city only once. The computational resources increase dramatically as the number of cities grows. This exponential increase in the number of possible paths denotes the NP-hard nature of the TSP[13].

[11] Md, Sandeeb. ",." , - YouTube, 21 October 2023, https://www.sciencedirect.com/topics/engineering/weight-function. Accessed 2 September 2023.
[12] Jain, Sandeep. "Find minimum weight cycle in an undirected graph." GeeksforGeeks, 21 February 2023, https://www.geeksforgeeks.org/find-minimum-weight-cycle-undirected-graph/. Accessed 10 September 2023.
[13] "Nondeterministic Polynomial Time Definition." DeepAI, https://deepai.org/machine-learning-glossary-and-terms/nondeterministic-polynomial-time. Accessed 13 September 2023.

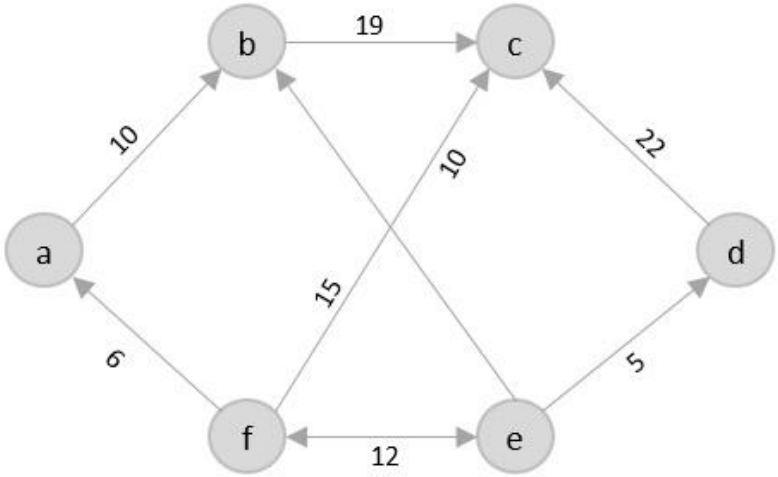


Figure 4: Example of Travelling Salesmen Problem[14]

NP-hard forms a class of complex challenges that are the most demanding puzzles in the field. "NP" means a set of decision issues that require polynomial time for the nondeterministic Turing machine to verify. NP-hard problems are known to have no known efficient solution algorithm that exists. If one finds an efficient solution to any NP-hard problem, it could be used to solve all the problems in the NP class efficiently which will result in a significant breakthrough in the field. Finding the best cost-effective path that links a given collection of cities and eventually gets back to the starting city is the objective of the Travelling Salesman Problem. Traveling Salesman Problem, which escalates in difficulty as the quantity of cities grows.[15] The challenge intensifies because the potential routes increase exponentially in tandem with the city count.

[14] "Travelling Salesman Problem (Greedy Approach)." Tutorialspoint, https://www.tutorialspoint.com/data_structures_algorithms/travelling_salesman_problem.htm. Accessed 20 February 2024.

[15] The Knapsack Problem, https://personal.utdallas.edu/~scniu/OPRE-6201/documents/DP3-Knapsack.pdf. Accessed 18 September 2023

3. Time complexity

Time complexity is a measure of the algorithm's computational requirements as a function of input size[16]. This helps in evaluating the algorithmic efficiency and estimating the rise in the runtime of an algorithm when more data is provided. To understand the feasibility and scalability of the implementation of algorithms in real-world applications the relationship between input size and runtime provides valuable insights for decision-making.

Input size can be defined as the number of elements in the input data[17]. Runtime is just the quantity of time required for an algorithm to execute[18]. This can be measured in various units like seconds, Millie seconds, or microseconds. For example, an algorithm has an input size of n. In that case, the runtime may be O(n), i.e., proportionate to the input's quantity[19].

2A.1 The Big O notation

For time complexity the central concept is the Big O notation algorithm's time complexity which can be represented using this method[20], which yields the algorithm's running time as a function of the algorithm's input size. This is a way of expressing the asymptotic behaviour of a function. The expression is as follows: $f(n) = O(g(n))$

Equation 1: Asymptotic behaviour of a function

Here f(n) represents the algorithm's resource usage and g (n) is a simpler function. g(n) and f(n) are asymptotically equal as n gets closer to infinity, Since Big O notation indicates the

[16] guide, Step. "Understanding Time Complexity with Examples - 2024." Great Learning, https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/. Accessed 19 May 2023

[17] "Time Complexity Analysis in Data Structures and Algorithms." EnjoyAlgorithms, https://www.enjoyalgorithms.com/blog/time-complexity-analysis-in-data-structure-and-algorithms. Accessed 19 May 2023.

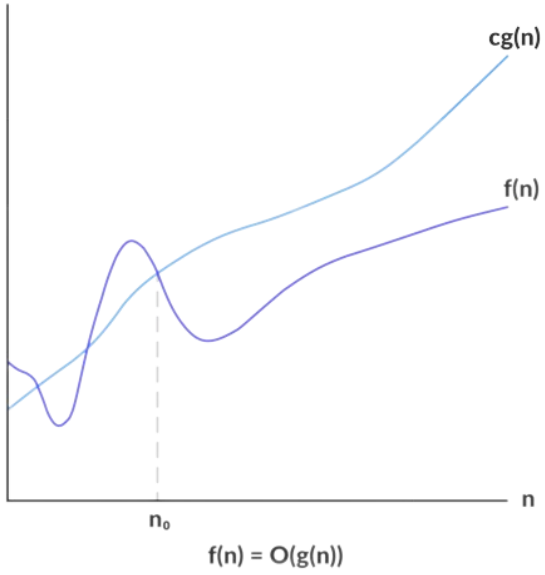
[18] "Running Time of Algorithms." HackerRank, https://www.hackerrank.com/challenges/runningtime/problem. Accessed 2 June 2023.

[19] Algorithmic Complexity, https://viterbi-web.usc.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html. 17 June 2023.

[20] Verma, Eshna. "Big O Notation in Data Structure: An Introduction." Simplilearn.com, 6 November 2023, https://www.simplilearn.com/big-o-notation-in-data-structure-article. Accessed 19 June 2023.

upper bound on the time complexity, it is frequently referred to as "worst-case time complexity"[21]. This indicates that it is the longest amount of time that may be predicted to be needed to run the algorithm[22]. The Big O notation's estimated value is never exceeded by the runtime.

The Big O notation's expression, f (n) = O(g(n)), can be seen in the graph below.



[23]

Figure 5: Graph displaying the worst-case time complexity and runtime

In the above depicted graph (Diagram 2), where the x-axis signifies input size and the y-axis represents runtime, it is evident that for input sizes exceeding a certain threshold, n_0, the function f(n) consistently becomes smaller than the product of g(n) and a constant, denoted as c. Thus, the existence of positive constants n_0 and c is established, validating the inequality

[21] Verma, Eshna. "Big O Notation in Data Structure: An Introduction." Simplilearn.com, 6 November 2023, https://www.simplilearn.com/big-o-notation-in-data-structure-article. Accessed 22 June 2023.
[22] guide, Step. "Understanding Time Complexity with Examples - 2024." Great Learning, https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/. Accessed 19 July 2023.
[23] Musharaf, Abdurrehman. "Asymptotic Notations. INTRODUCTION: | by Abdurrehman Musharaf | Oct, 2023." Medium, 31 October 2023, https://medium.com/@abdurrehman.musharaf/asymptotic-notations-6fb9e7b7293e. Accessed 1 August 2023.

$0 \leq f(n) \leq cg(n)$ for all $n \geq n_o$[24].It's crucial to note that these constants, c, and n_o, remain unaltered regardless of the input size, ensuring a consistent relationship. In this case, the input size is indicated by n, and the minimum value of n that ensures f(n) cg(n). Generally, the term with the highest order in f(n) corresponds to g(n), exerting the most significant impact on the expression as n increases. To illustrate, consider the function f(n) = $7n^2$ + 3n + 1, where g(n) is n^2. In this case, the big O notation of the expression is O (n^2).

The time complexity of the algorithms can be evaluated using the following two big O notation properties:

Equation 2: Big O notation properties

Property 1: $O(f(n)) + O(g(n)) = O(f(n)) + g(n))$

Property 2: $O(f(n)) * O(g(n)) = O(f(n)) * g(n))$

[25]

When we look at shortest-path algorithms, time complexity helps in selecting the efficient method for finding the best possible optimal routes in graphs.

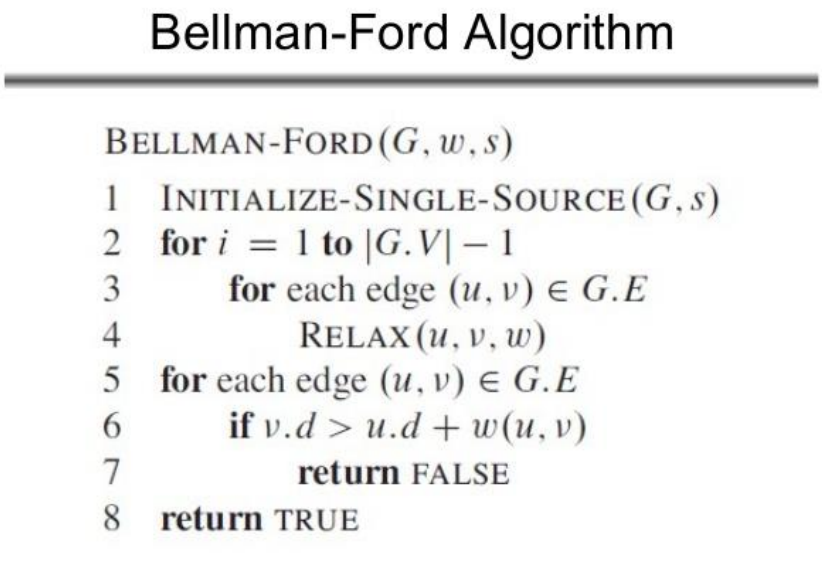
In the pursuit of analysing the time complexity of shortest-path algorithms, this investigation maintains focus by exclusively utilizing directed and positively weighted graphs. To ensure clarity, it's important to provide a concise overview of how the calculation of the shortest path on these graphs is approached.

[24] Jain, Sandeep. "Properties of Asymptotic Notations." GeeksforGeeks, 14 June 2023, https://www.geeksforgeeks.org/properties-of-asymptotic-notations/. Accessed 2 August 2023.
[25] Jain, Sandeep. "Properties of Asymptotic Notations." GeeksforGeeks, 14 June 2023, https://www.geeksforgeeks.org/properties-of-asymptotic-notations/. Accessed 10 August 2023.

4. Algorithms

4A. Bellman-ford algorithm

The Bellman-ford algorithm is ideal for graphs with directed edges and weighted connections; this makes it suitable for the scenario where the edge weight can be negative. Here we can use the bell-man Ford to calculate the shortest path for various instances in a traveling salesman problem. When presented with negative edge weights, the algorithm might need to explore more paths leading to a longer runtime than predicted by Time complexity.



[26]

Figure 6: The Pseudocode of Bellman ford algorithm

As a difference to Dijkstra's algorithm, the Bellman-Ford algorithm isn't classified as a greedy algorithm. It stands out by systematically relaxing every edge in the graph, ensuring that the solution obtained is the optimal global solution. This meticulous methodology guarantees an exhaustive exploration of all potential paths. However, this exhaustive exploration is at the cost

[26] "Negative cycle detection using Bellman-Ford and its correctness." Computer Science Stack Exchange, 9 June 2020, https://cs.stackexchange.com/questions/126945/negative-cycle-detection-using-bellman-ford-and-its-correctness. Accessed 26 September 2023.

of increased runtime compared to Dijkstra's algorithm. Even though both algorithms arrive at the same solution for shortest path problems, the Bellman-Ford algorithm takes longer due to its thorough examination of all edges and paths within the graph.

4B. Dijkstra's Algorithm

When we look at Dijkstra's Algorithm it is tailored to work in situations where edge weights are non-negative. It prioritizes vertices with the smallest tentative distances and as it expands it iterates to neighbouring vertices[27]. When we apply Dijkstra's Algorithm to TSP instances with non-negative edge weights the actual runtime is expected to closely align with the calculated time complexity. Functioning as a greedy algorithm, it consistently opts for the optimal solution at each stage without delving into subproblems. Using this method, the edge with the least weight is determined by relaxing each edge that leaves the current vertex. It successfully finds all of the shortest paths from its source vertex to the other vertex in the graph through a series of iterative processes. Though greedy algorithms may not be universally applicable due to their lack of comprehensive problem consideration, they excel in solving shortest-path problems. This is explained by the shortest path's essential feature, which states that every path that links two vertices must also contain other shortest paths. The resulting path remains the shortest even after the greedy algorithm adds individual small sub paths. Every vertex is kept in a minimal-priority queue in order to continuously update the minimum value of the shortest path estimate for each vertex. Each vertex's $v.d$ serves as the key in this queue. Dijkstra's algorithm facilitates comparisons between the shortest path estimates of each vertex, enabling updates when a lower value is discovered.

[27] "How Dijkstra's Algorithm works." Programiz, https://www.programiz.com/dsa/dijkstra-algorithm. Accessed 15 October 2023.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

[28]

Figure 7: The Pseudocode of Dijkstra's algorithm

4B.1 Calculating the theoretical time complexity

Bellman-Ford initializes the graph (O(V)) first, after which a loop iterates V - 1 instances, relaxing the graph ((V-1) × O(E)) for each iteration. The next step involves iterating through each edge again and determining whether a relation could still arise (O(E)) to determine whether a negative-weight cycle exists.

[28] "How to get the shortest path from s to t in a graph $G_1(V,E_1)$ after using Dijkstra's algorithm?" Math Stack Exchange, 5 January 2019, https://math.stackexchange.com/questions/3062775/how-to-get-the-shortest-path-from-s-to-t-in-a-graph-g-1v-e-1-after-using. Accessed 3 November 2023.

$$O(V) + (V\text{-}1) \times O(E) + O(E)$$

$$= O(V + (V\text{-}1)E + E)$$

$$= O(V + VE)$$

$$= O(VE)$$

Figure 8: Time complexity of the algorithm

Dijkstra sets the graph's initial state (O(V)). S and Q are then initialized (O (1)). Then, iterating over all of the vertices (O(V)), it relaxes all of the edges that leave the vertex once (O(E)), extracting a vertex from Q and adding it to S (V × O (EXTRACT - MIN))[29].

The min-priority queue's implementation affects the temporal complexity of the EXTRACT - MIN operation is O (lg v), where lg v = log 2v, if the queue is built effectively. This is the case in scenario 1.

[29] "Dijkstra's algorithm finds a shortest path from a source vertex." Dartmouth CS, https://www.cs.dartmouth.edu/~thc/cs10/lectures/0509/0509.html. Accessed 5 November 2023.

$$O(V) + O(1) + V \times O(LgV) + O(E)$$

$$= O(V + 1 + VLgV + E)$$

$$= O(V(1 + lgV) + E)$$

$$= O(Vlg\ V + E)$$

Figure 9: Time complexity of Dijkstra's algorithm

5. Comparison

5A. Primary investigation

We examined the time complexities of the Dijkstra and Bellman-Ford algorithms in detail in the section above, and this demonstrated how dependent they are on the variables V (number of vertices) and E (number of edges). Here the research focuses on understanding the variations in runtime concerning changes in input size. In the context of this, it's important to streamline the analysis by concentrating solely on the variable V. By delving deep into the relationship between these quantities, the goal is to make it easier to compare the time complexities of these algorithms using V as a reference point.

$$\text{Edge density} = \frac{\text{Number of edges in the graph}}{\text{Maximum possible number of edges}}$$

Equation 3: Edge density equation

Utilizing the equation given above, we determine the upper limit for the number of edges relative to the number of vertices (V) in a graph. V(V-1)/2 is the result of calculating the combination of two vertices in the calculation.

Here the relation between E and V is as follows,

$$0.2 = \frac{Number\ of\ edges\ int\ he\ graph}{Maximum\ number\ of\ edges} = \frac{E}{\frac{V(v-1)}{2}}$$ [30]

Equation 4: 1 - Relation of E and V

$$E = 0.2\left(\frac{V(v-1)}{2}\right) = 0.1(V^2 - V)$$

Equation 5: 2 - Relation of E and V

The time complexities of the Bellman-Ford and Dijkstra's algorithms are expressed in terms of both V (vertices) and E (edges), as we have previously discussed.

5B. Data collection

For the Traveling Salesman Problem (TSP) investigation, I will leverage real-world data encompassing the distances between various capital cities in India This dataset will be instrumental in simulating the TSP's complexities within a geographic context (Appendix 10C). Graph theory and network analysis-specific Python package NetworkX is used to study and visualise this dataset.

Using the two applications NetworkX and Matplotlib, the Hamiltonian path was generated for the dataset crested, containing the distances between India's capitals. The programme that was

[30] https://math.stackexchange.com/questions/1526372/what-is-the-definition-of-the-density-of-a-graph#:~:text=Note%20that%20the%20maximum%20number,V%7C%E2%88%921)2.&text=Yeah!,%7C*%7Cv%2D1%7C%2F2. Accessed 10 November 2023.

run was intended to examine the scalability and efficiency of two basic graph algorithms, Bellman-Ford and Dijkstra's, utilising datasets that were representative of Hamiltonian maps and Travelling Salesman Problems (TSP). The datasets that served as the foundation for the graph constructions contained a variety of cities and their respective distances.

The structure of the collection is unique and reflects real-world geographical issues. It represented the graph's representation in a weighted and directed manner by using the distances between many cities as edges and the cities themselves as vertices. There was a complex network of connections spanning 27 cities in total.

The dataset's variable edge density was one of its key characteristics. Certain cities were situated in closer proximity to one another, leading to a denser subgraph in specific areas. Cities in the same geographic region, for instance, had more edges, but cities in distant regions had less connections. This mimics the reality of distances between cities and is a significant factor that impacted the algorithm runtimes.

Implemented Bellman-Ford and Dijkstra's algorithms to determine the shortest routes between Delhi, the nation's centre, and every other city. A key objective was to compare the actual runtimes of both algorithms on various instances derived from the dataset. The results showed that the actual runtimes of Dijkstra's algorithm were consistently shorter than Bellman-Ford's.

6. Result Analysis

The collected data is placed in Section 10 B of Appendix in the form of tables for convenience. This discussion is based on the results with reference Appendix 10B. The actual runtime results for the Bellman-Ford and Dijkstra algorithms are consistent with their previously calculated

time complexity, The time complexity of Bellman-Ford's algorithm is O (|V||E|)[31], whereas that of Dijkstra's algorithm is O (|V| + |E|log|V|)[32]. This means that Dijkstra's algorithm is expected to be faster than Bellman-Ford's algorithm for problems with a large number of edges, which is the case for the Indian road network.

In the experiment to calculate the shortest path between Delhi to considering all the capitals of India, Dijkstra's algorithm took 0.004556179046630859 seconds to find the shortest route, while Bellman-Ford's algorithm took 0.006062507629394531 seconds. For a simple problem like this one, the runtime difference is relatively small, but for larger problems, it would be more significant. The reason for the findings is that an algorithm's real runtime depends on a number of factors, including the method's implementation, the programming language being used, and the hardware the algorithm is executing on. These factors are in addition to the algorithm's time complexity.

This experiment used one computer to run the python program of the Bellman-Ford and Dijkstra algorithms. Dijkstra algorithm is typically more scalable and efficient compared to Bellman-Ford algorithm This is due to the fact that Dijkstra's approach is less prone to negative edge weights and has a lower time complexity[33].

It's also critical to consider instances in which the actual run time and the theoretical time complexity might not align. However, Dijkstra's method typically outperforms Bellman's algorithm. This pattern may not always hold in situations when there are many edges and no negative edge weights. When the graph has negative edge weights, this is one example. In this

[31] "Bellman-Ford Algorithm - javatpoint." Javatpoint, https://www.javatpoint.com/bellman-ford-algorithm. Accessed 22 November 2023.

[32] "Dijkstra's Algorithm Runtime." EECS: www-inst.eecs.berkeley.edu, https://inst.eecs.berkeley.edu/~cs61bl/r//cur/graphs/dijkstra-algorithm-runtime.html?topic=lab24.topic&step=4&course=. Accessed 24 November 2023.

[33] Mukhlif, Fadhil. "(PDF) Comparative Study On Bellman-Ford And Dijkstra Algorithms." ResearchGate, 20 April 2020, https://www.researchgate.net/publication/340790429_Comparative_Study_On_Bellman-Ford_And_Dijkstra_Algorithms. Accessed 28 November 2023.

case, Bellman-Ford's algorithm might outperform Dijkstra's method even if it has a higher time complexity. Moreover, in cases where the groan is sparse, meaning when there are relatively few edges in comparison to number of vertices, Bellman-Ford's algorithm may potentially show faster or even better runtime performance in these situations. This is because. Unlike Dijkstra's algorithm, which explores every vertex based on the distance from the source node, Bellman-Ford's algorithm explores all edges in each iteration, making it less impacted due to the graph sparsity.

Even when considering the complexities of real-world road and networks, the runtime of both algorithms can be greatly impacted by variables like traffic conditions, road closures, and congestion. Dijkstra's algorithm may experience overhead in maintaining its priority queue when the shortest path needs to be recalculated frequently due to the dynamic changes in the environment. This puts Dijkstra's algorithm at a disadvantage, decrease in performance when compared to Bellman-Ford algorithm, which recalculates paths more simply. Hence, for a reduced time complexity it is ideally to go with Dijkstra's algorithm. But, in the case of negative weights Bellman-Ford algorithm is efficient and ideal[34].

7. Conclusion

The research question, ***"To what extent does Dijkstra's algorithm outperform the Bellman-Ford algorithm in terms of time complexity while solving the Traveling Salesman Problem as a special case of the Hamiltonian Path in the Complex graph search category?"***, has successfully answered by finding that Dijkstra's complimented by its less calculated Time complexity is more efficient than Bellman-Ford. In the context to India's capital cities, by utilizing the NetworkX in python, a dataset representing the map of India where each node

[34] Mukhlif, Fadhil. "(PDF) Comparative Study On Bellman-Ford And Dijkstra Algorithms." ResearchGate, 20 April 2020, https://www.researchgate.net/publication/340790429_Comparative_Study_On_Bellman-Ford_And_Dijkstra_Algorithms. Accessed 1 December 2023.

represented a capital city. Executing the chosen algorithms on the dataset gave valuable insights into their real-world performance. By considering recorded execution times and comparing it with the calculated time complexity, it provided an understanding of algorithmic efficiency.

The biggest limitation of this evaluation was the assumption of uniform road conditions and the absence of real-time traffic data. Findings may be limited by the fact that the dataset covering the capital cities of India may not accurately reflect the intricacies of actual road networks. Furthermore, in order to avoid taking into account dynamic elements like traffic congestion and road closures, which could affect algorithm runtime in real-world applications, this analysis assumed uniform road conditions.

To address these limitations and enhance the reliability of the research in the real world, several improvements can be made. For example, exploring diverse road networks, expanding the data set to accommodate additional factors like road connectivity and traffic patterns can provide with a more realistic representation of India's road's network. Moreover, carrying out sensitivity analysis to evaluate the algorithms robustness of the algorithms to varying parameters and environmental circumstances would augment the dependability of the results.

In conclusion, this study demonstrates the value of using the Dijkstra and Bellman-Ford algorithms to solve Travelling Sales Men and Hamiltonian Path Problems for the capital cities of India. Although our results shed light on the efficiency of the algorithm in this particular situation, more investigation and improvements are necessary to overcome the drawbacks and progress the field of graph optimisation, routing for practical uses. It is important to also note that the real-world scenarios may present deviations from theoretical expectations. Bellman-Ford's showcases competitive performance in sparse graph and negative edge weights showcasing its adaptability to a wider range of graph properties. The actual runtime of an

algorithm adheres to the calculation of its time complexity to some extent, but other factors such as the implementation of the algorithm, the programming language used, and the hardware on which the algorithm is running can also have a significant impact on runtime.

So, the Research Question can be furthered to see to what extent does Bellman-Ford outperform Dijkstra's algorithm on a real-life Hamiltonian path in finding a Hamiltonian path in real-life scenarios, considering all complex factors such as blocked roads, congestions, and non-uniform datasets with negative weights, utilizing real-life traffic data?

8. Bibliography

8A. Books

- Cormen, Thomas H. *Algorithms Unlocked*. Cambridge, MA, 2013.

- Roughgarden, Tim. *Algorithms Illuminated: Graph algorithms and data structures. Part 2*. Soundlikeyourself Publishing LLC, 2018

- Cook, William. In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation. Princeton University Press, 2012.

8B. Websites

- Nikolopoulou, Kassiani. “What Is an Algorithm? | Definition & Examples.” Scribbr, 9 August 2023, https://www.scribbr.com/ai-tools/what-is-an-algorithm/.

- Kharwal, Aman. “Worst Case, Average Case, and Best Case | Aman Kharwal.” thecleverprogrammer, 9 November 2020, https://thecleverprogrammer.com/2020/11/09/worst-case-average-case-and-best-case/.

- Wikipedia, https://adacomputerscience.org/concepts/struct_graph?examBoard=all&stage=all.

- Md, Sandeeb. “,.” , - YouTube, 21 October 2023, https://link.springer.com/article/10.1007/s10703-019-00337-w.

- Jain, Sandeep. “What are the differences between Bellman Ford's and Dijkstra's algorithms?” GeeksforGeeks, 23 June 2022, https://www.geeksforgeeks.org/what-are-the-differences-between-bellman-fords-and-dijkstras-algorithms/.

- guide, Step. “Understanding Time Complexity with Examples - 2024.” Great Learning, https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/.

- Time Complexity Analysis in Data Structures and Algorithms.” EnjoyAlgorithms, https://www.enjoyalgorithms.com/blog/time-complexity-analysis-in-data-structure-and-algorithms.

- “Running Time of Algorithms.” HackerRank, https://www.hackerrank.com/challenges/runningtime/problem.

- Algorithmic Complexity, https://viterbi-web.usc.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html.

- Verma, Eshna. “Big O Notation in Data Structure: An Introduction.” Simplilearn.com, 6 November 2023, https://www.simplilearn.com/big-o-notation-in-data-structure-article.- Verma, Eshna. “Big O Notation in Data Structure: An Introduction.” Simplilearn.com, 6 November 2023, https://www.simplilearn.com/big-o-notation-in-data-structure-article.

- guide, Step. “Understanding Time Complexity with Examples - 2024.” Great Learning, https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/.

- Jain, Sandeep. “Properties of Asymptotic Notations.” GeeksforGeeks, 14 June 2023, https://www.geeksforgeeks.org/properties-of-asymptotic-notations/.

- Jain, Sandeep. “Properties of Asymptotic Notations.” GeeksforGeeks, 14 June 2023, https://www.geeksforgeeks.org/properties-of-asymptotic-notations/.

- “Graphs - Intro to Data Structures.” Codology, https://www.codology.org/intro-to-data-structures/graphs.

- Md, Sandeeb. “,.” , - YouTube, 21 October 2023, https://www.sciencedirect.com/topics/mathematics/weighted-graph.

- “Directed and Undirected Graphs - MATLAB & Simulink.” MathWorks, https://www.mathworks.com/help/matlab/math/directed-and-undirected-graphs.html.

- Md, Sandeeb. “,.” , - YouTube, 21 October 2023, https://www.sciencedirect.com/topics/engineering/weight-function.

- Jain, Sandeep. “Find minimum weight cycle in an undirected graph.” GeeksforGeeks, 21 February 2023, https://www.geeksforgeeks.org/find-minimum-weight-cycle-undirected-graph/.

- “Nondeterministic Polynomial Time Definition.” DeepAI, https://deepai.org/machine-learning-glossary-and-terms/nondeterministic-polynomial-time.

- The Knapsack Problem, https://personal.utdallas.edu/~scniu/OPRE-6201/documents/DP3-Knapsack.pdf.

- Jaimini, Vaibhav. “Hamiltonian Path Tutorials & Notes | Algorithms.” HackerEarth, https://www.hackerearth.com/practice/algorithms/graphs/hamiltonian-path/tutorial/.

- “How to get the shortest path from s to t in a graph $G_1(V,E_1)$ after using Dijkstra's algorithm?” Math Stack Exchange, 5 January 2019, https://math.stackexchange.com/questions/3062775/how-to-get-the-shortest-path-from-s-to-t-in-a-graph-g-1v-e-1-after-using.

- “Dijkstra's algorithm finds a shortest path from a source vertex.” Dartmouth CS, https://www.cs.dartmouth.edu/~thc/cs10/lectures/0509/0509.html. Accessed 29 January 2024.

- “Bellman-Ford Algorithm - javatpoint.” Javatpoint, https://www.javatpoint.com/bellman-ford-algorithm.

- “Dijkstra's Algorithm Runtime.” EECS: www-inst.eecs.berkeley.edu, https://inst.eecs.berkeley.edu/~cs61bl/r//cur/graphs/dijkstra-algorithm-runtime.html?topic=lab24.topic&step=4&course=.

- Mukhlif, Fadhil. “(PDF) Comparative Study On Bellman-Ford And Dijkstra Algorithms.” ResearchGate, 20 April 2020,

https://www.researchgate.net/publication/340790429_Comparative_Study_On_Bellman-Ford_And_Dijkstra_Algorithms.

- ". . - definition of . . by The Free Dictionary." The Free Dictionary, https://iopscience.iop.org/article/10.1088/1757-899X/917/1/012077.

- Algorithms and Data Structures - Complexity of Algorithms, https://users.pja.edu.pl/~msyd/wyka-eng/complexity2.pdf.

- Kuo, Marc. "Algorithms for the Travelling Salesman Problem." Routific, 8 November 2023, https://www.routific.com/blog/travelling-salesman-problem.

- "%LJ 2 QRWDWLRQ %LJ 2 QRWDWLRQ Big O notation (with a capital letter O, not a zero), also called Landau's symbol, is a symbolism." MIT, https://web.mit.edu/16.070/www/lecture/big_o.pdf.

- Dham, Mayank. "Asymptotic Notation: Big Oh, Omega and Theta Notation." PrepBytes, 27 January 2023, https://www.prepbytes.com/blog/miscellaneous/asymptotic-notation-types-and-uses/.

- ., Brian P. "Runtime Definition - What does runtime mean?" TechTerms.com, 6 April 2023, https://techterms.com/definition/runtime.

- Mukhlif, Fadhil. "(PDF) Comparative Study On Bellman-Ford And Dijkstra Algorithms." ResearchGate, 20 April 2020, https://www.researchgate.net/publication/340790429_Comparative_Study_On_Bellman-Ford_And_Dijkstra_Algorithms.

- "[1301.3120] An edge density definition of overlapping and weighted graph communities." arXiv, 14 January 2013, https://arxiv.org/abs/1301.3120.

- Jain, Sandeep. "What is Dijkstra's Algorithm? | Introduction to Dijkstra's Shortest Path Algorithm." GeeksforGeeks, https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/.

8C. Images

- Dijkstra's algorithm finds a shortest path from a source vertex." Dartmouth CS, https://www.cs.dartmouth.edu/~thc/cs10/lectures/0509/0509.html.

- "Negative cycle detection using Bellman-Ford and its correctness." Computer Science Stack Exchange, 9 June 2020, https://cs.stackexchange.com/questions/126945/negative-cycle-detection-using-bellman-ford-and-its-correctness.

- Travelling Salesman Problem (Greedy Approach)." Tutorialspoint, https://www.tutorialspoint.com/data_structures_algorithms/travelling_salesman_problem.htm. Accessed 20 February 2024.

- "Negative cycle detection using Bellman-Ford and its correctness." Computer Science Stack Exchange, 9 June 2020, https://cs.stackexchange.com/questions/126945/negative-cycle-detection-using-bellman-ford-and-its-correctness.

- Rössel, Johannes. "File:Directed acyclic graph 2.svg." Wikipedia, https://en.m.wikipedia.org/wiki/File:Directed_acyclic_graph_2.svg.

- Young, Michael. "Graph Theory." The Computer Science Handbook, https://www.thecshandbook.com/Graph_Theory.

- Jain, Sandeep. “Properties of Asymptotic Notations.” GeeksforGeeks, 14 June 2023, https://www.geeksforgeeks.org/properties-of-asymptotic-notations/.

- Musharaf, Abdurrehman. “Asymptotic Notations. INTRODUCTION: | by Abdurrehman Musharaf | Oct, 2023.” Medium, 31 October 2023, https://medium.com/@abdurrehman.musharaf/asymptotic-notations-6fb9e7b7293e.

9. Glossary

9A. Terminologies

1. Time Complexity: This describes the amount of time required to execute an algorithm, its also a type of computational complexity. [35]

2. Travelling Salesman Problem: This is a type of challenge which is about finding the shortest path or shortest route for a salesperson and is used, given a starting point, the number of cities (nodes), and an ending point. [36]

3. The Big O Notation: This is known as the upper bound of the time complexity.[37]

4. Asymptotic Notations: These are mathematical tools used for studying the behaviour of an algorithm considering the input provided. [38]

5. Runtime: Describes the period of time during which a computer program is running.[39]

6. Edge Density: Edge Density is the number of edges divided by the maximal umber of edges. [40]

7. Bellman Ford Algorithm: It is the single source shortest path algorithm.

[35] Algorithms and Data Structures - Complexity of Algorithms, https://users.pja.edu.pl/~msyd/wyka-eng/complexity2.pdf. Accessed 1 February 2024.
[36] Kuo, Marc. “Algorithms for the Travelling Salesman Problem.” Routific, 8 November 2023, https://www.routific.com/blog/travelling-salesman-problem. Accessed 1 February 2024.
[37] “%LJ 2 QRWDWLRQ %LJ 2 QRWDWLRQ Big O notation (with a capital letter O, not a zero), also called Landau's symbol, is a symbolism.” MIT, https://web.mit.edu/16.070/www/lecture/big_o.pdf. Accessed 5 February 2024.
[38] Dham, Mayank. “Asymptotic Notation: Big Oh, Omega and Theta Notation.” PrepBytes, 27 January 2023, https://www.prepbytes.com/blog/miscellaneous/asymptotic-notation-types-and-uses/. Accessed 5 February 2024.
[39] ., Brian P. “Runtime Definition - What does runtime mean?” TechTerms.com, 6 April 2023, https://techterms.com/definition/runtime. Accessed 5 February 2024.
[40] “[1301.3120] An edge density definition of overlapping and weighted graph communities.” arXiv, 14 January 2013, https://arxiv.org/abs/1301.3120. Accessed 5 February 2024.

8. Dijkstra Algorithm: It is capable of solving many single-source shortest path problems which have non-negative edge weight in graphs.[41]

[41] Jain, Sandeep. “What is Dijkstra’s Algorithm? | Introduction to Dijkstra's Shortest Path Algorithm.” GeeksforGeeks, https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/. Accessed 5 February 2024.

10. Appendices

10A. Specifications of the computer system

Computer system model: HP Envy 33.8 cm x360 Laptop 13-bf0121TU

CPU: AMD Ryzen 5 5600U with Radeon Graphics, 2.30 GHz

OS: Microsoft Windows 11 Enterprise

10B. Table of Raw Data

Data Table 1

City	Distance from Delhi in Kilometres
Amaravati	1,204 km
Chandigarh	244 km
Dehradun	245 km
Gandhinagar	970 km
Gangtok	1,605 km
Hyderabad	1,573 km
Itanagar	2,210 km
Jaipur	306 km
Shimla	343 km
Ranchi	1,306 km
Bengaluru	2,105 km
Thiruvananthapuram	2,967 km
Bhopal	787 km
Mumbai	1,416 km
Imphal	2,424 km
Shillong	2,010 km
Aizawl	2,272 km
Bhubaneswar	1,624 km
Chennai	2,265 km
Kohima	2,300 km
Lucknow	550 km
Panaji	1,900 km
Patna	1,055 km
Agartala	2,342 km
Kolkata	1,472 km
Raipur	1,200 km
Dispur	1,955 km

Data Table 2

City	Distance from Amaravati in Kilometres
Chandigarh	2114
Dehradun	2066
Gandhinagar	1500
Gangtok	1900
Hyderabad	280
Itanagar	2450
Jaipur	1730
Shimla	2230
Ranchi	1302
Bengaluru	665
Thiruvananthapuram	1219
Bhopal	1130
Mumbai	1000
Imphal	2700
Shillong	2282
Aizawl	2650
Bhubaneswar	793
Chennai	449
Kohima	2534
Lucknow	1620
Panaji	969
Patna	1493
Agartala	2719
Kolkata	1233
Raipur	751
Dispur	2196

Data Table 3

City	Distance from Chandigarh in Kilometres
Dehradun	171
Gandhinagar	1105
Gangtok	1858
Hyderabad	1832
Itanagar	2431
Jaipur	482
Shimla	113
Ranchi	1559
Bengaluru	2424
Thiruvananthapuram	3160

Bhopal	1007
Mumbai	1634
Imphal	2654
Shillong	2269
Aizawl	2638
Bhubaneswar	2041
Chennai	2458
Kohima	2522
Lucknow	779
Panaji	2103
Patna	1279
Agartala	2707
Kolkata	1812
Raipur	1509
Dispur	2184

Data Table 4

City	Distance from Dehradun in Kilometres
Gandhinagar	1174
Gangtok	1545
Hyderabad	1786
Itanagar	2107
Jaipur	556
Shimla	223
Ranchi	1512
Bengaluru	2379
Thiruvananthapuram	3110
Bhopal	993
Mumbai	1689
Imphal	2598
Shillong	1958
Aizawl	2325
Bhubaneswar	1994
Chennai	2413
Kohima	2466
Lucknow	751
Panaji	2145
Patna	1257
Agartala	2651
Kolkata	1764
Raipur	1462
Dispur	2128

Data Table 5

City	Distance from Gandhinagar in Kilometres
Gangtok	2246
Hyderabad	1207
Itanagar	2805
Jaipur	634
Shimla	1204
Ranchi	1611
Bengaluru	1515
Thiruvananthapuram	2264
Bhopal	598
Mumbai	545
Imphal	3033
Shillong	2648
Aizawl	3017
Bhubaneswar	1711
Chennai	1845
Kohima	2901
Lucknow	1186
Panaji	1124
Patna	1692
Agartala	3086
Kolkata	2039
Raipur	1147
Dispur	2563

Data Table 6

City	Distance from Gangtok in Kilometres
Hyderabad	2024
Itanagar	766
Jaipur	1630
Shimla	1939
Ranchi	793
Bengaluru	2616
Thiruvananthapuram	3071
Bhopal	1559
Mumbai	2293
Imphal	994
Shillong	609
Aizawl	978
Bhubaneswar	1075

Chennai	2301
Kohima	862
Lucknow	1058
Panaji	2629
Patna	571
Agartala	1070
Kolkata	696
Raipur	1301
Dispur	524

Data Table 7

City	Distance from Hyderabad in Kilometres
Itanagar	2585
Jaipur	1445
Shimla	1912
Ranchi	1302
Bengaluru	576
Thiruvananthapuram	1307
Bhopal	851
Mumbai	708
Imphal	2813
Shillong	2429
Aizawl	2798
Bhubaneswar	1045
Chennai	627
Kohima	2681
Lucknow	1341
Panaji	721
Patna	1473
Agartala	2866
Kolkata	1486
Raipur	714
Dispur	2343

Data Table 8

City	Distance from Itanagar in Kilometres
Jaipur	2201
Shimla	2508
Ranchi	1361
Bengaluru	3186
Thiruvananthapuram	3639

Bhopal	2129
Mumbai	2864
Imphal	573
Shillong	393
Aizawl	668
Bhubaneswar	1643
Chennai	2869
Kohima	441
Lucknow	1628
Panaji	3199
Patna	1142
Agartala	758
Kolkata	1264
Raipur	1871
Dispur	325

Data Table 9

City	Distance from Jaipur in Kilometres
Shimla	574
Ranchi	1331
Bengaluru	1924
Thiruvananthapuram	2672
Bhopal	598
Mumbai	1180
Imphal	2414
Shillong	2030
Aizawl	2399
Bhubaneswar	1824
Chennai	2074
Kohima	2282
Lucknow	573
Panaji	1636
Patna	1074
Agartala	2467
Kolkata	1581
Raipur	1164
Dispur	1944

Data Table 10

City	Distance from Shimla in Kilometres
Ranchi	1562
Bengaluru	2499
Thiruvananthapuram	3231
Bhopal	1213
Mumbai	1794
Imphal	2719
Shillong	2334
Aizawl	2703
Bhubaneswar	2045
Chennai	2533
Kohima	2586
Lucknow	907
Panaji	2250
Patna	1409
Agartala	2771
Kolkata	1815
Raipur	1517
Dispur	2249

Data Table 11

City	Distance from Ranchi in Kilometres
Bengaluru	1939
Thiruvananthapuram	2506
Bhopal	1013
Mumbai	1637
Imphal	1583
Shillong	1198
Aizawl	1567
Bhubaneswar	510
Chennai	1736
Kohima	1451
Lucknow	762
Panaji	1972
Patna	338
Agartala	1635
Kolkata	402
Raipur	589
Dispur	1113

Data Table 12

City	Distance from Bengaluru in Kilometres
Thiruvananthapuram	837
Bhopal	1442
Mumbai	982
Imphal	3278
Shillong	2893
Aizawl	3262
Bhubaneswar	1428
Chennai	346
Kohima	3146
Lucknow	1932
Panaji	595
Patna	2064
Agartala	3457
Kolkata	1869
Raipur	1358
Dispur	2808

Data Table 13

City	Distance from Thiruvananthapuram in Kilometres
Bhopal	2167
Mumbai	1610
Imphal	3847
Shillong	3462
Aizawl	3831
Bhubaneswar	1997
Chennai	781
Kohima	3715
Lucknow	2657
Panaji	967
Patna	2812
Agartala	3900
Kolkata	2439
Raipur	1965
Dispur	3377

Data Table 14

City	Distance from Bhopal in Kilometres

Mumbai	775
Imphal	2347
Shillong	1962
Aizawl	2331
Bhubaneswar	1183
Chennai	1477
Kohima	2214
Lucknow	645
Panaji	1320
Patna	1006
Agartala	2399
Kolkata	1430
Raipur	628
Dispur	1876

Data Table 14

City	Distance from Mumbai
Imphal	3082
Shillong	2697
Aizawl	3066
Bhubaneswar	1605
Chennai	1317
Kohima	2950
Lucknow	1446
Panaji	569
Patna	1741
Agartala	3134
Kolkata	1886
Raipur	1050
Dispur	2612

Data Table 15

City	Distance from Imphal in Kilometres
Shillong	546
Aizawl	396
Bhubaneswar	1893
Chennai	3118
Kohima	137
Lucknow	1877
Panaji	3448
Patna	1391

Agartala	540
Kolkata	1514
Raipur	2120
Dispur	478

Data Table 16

City	Distance from Shillong in Kilometres
Aizawl	374
Bhubaneswar	1506
Chennai	2732
Kohima	412
Lucknow	1491
Panaji	3062
Patna	1004
Agartala	443
Kolkata	1127
Raipur	1734
Dispur	91

Data Table 17

City	Distance from Aizawl in Kilometres
Bhubaneswar	1875
Chennai	3101
Kohima	526
Lucknow	1860
Panaji	3431
Patna	1374
Agartala	322
Kolkata	1496
Raipur	2103
Dispur	460

Data Table 18

City	Distance from Bhubaneswar in Kilometres
Chennai	1227
Kohima	1718
Lucknow	1240
Panaji	1791
Patna	816
Agartala	1903

Kolkata	442
Raipur	558
Dispur	1381

Data Table 19

City	Distance from Chennai in Kilometres
Kohima	2942
Lucknow	1964
Panaji	949
Patna	2040
Agartala	3127
Kolkata	1666
Raipur	1191
Dispur	2604

Data Table 20

City	Distance from Kohima in Kilometres
Lucknow	1744
Panaji	3315
Patna	1257
Agartala	618
Kolkata	1380
Raipur	1987
Dispur	344

Data Table 21

City	Distance from Lucknow in Kilometres
Panaji	1942
Patna	502
Agartala	1896
Kolkata	1009
Raipur	782
Dispur	1373

Data Table 22

City	Distance from Panaji in Kilometres
Patna	2087
Agartala	3480
Kolkata	2232

Raipur	1396
Dispur	2957

Data Table 23

City	Distance from Patna in Kilometres
Agartala	1425
Kolkata	553
Raipur	732
Dispur	903

Data Table 24

City	Distance from Agartala in Kilometres
Kolkata	1565
Raipur	2171
Dispur	529

Data Table 25

City	Distance from Kolkata in Kilometres
Raipur	834
Dispur	1011

Data Table 26

City	Distance from Raipur in Kilometres
Dispur	1594

10.C Image of the generated Graph

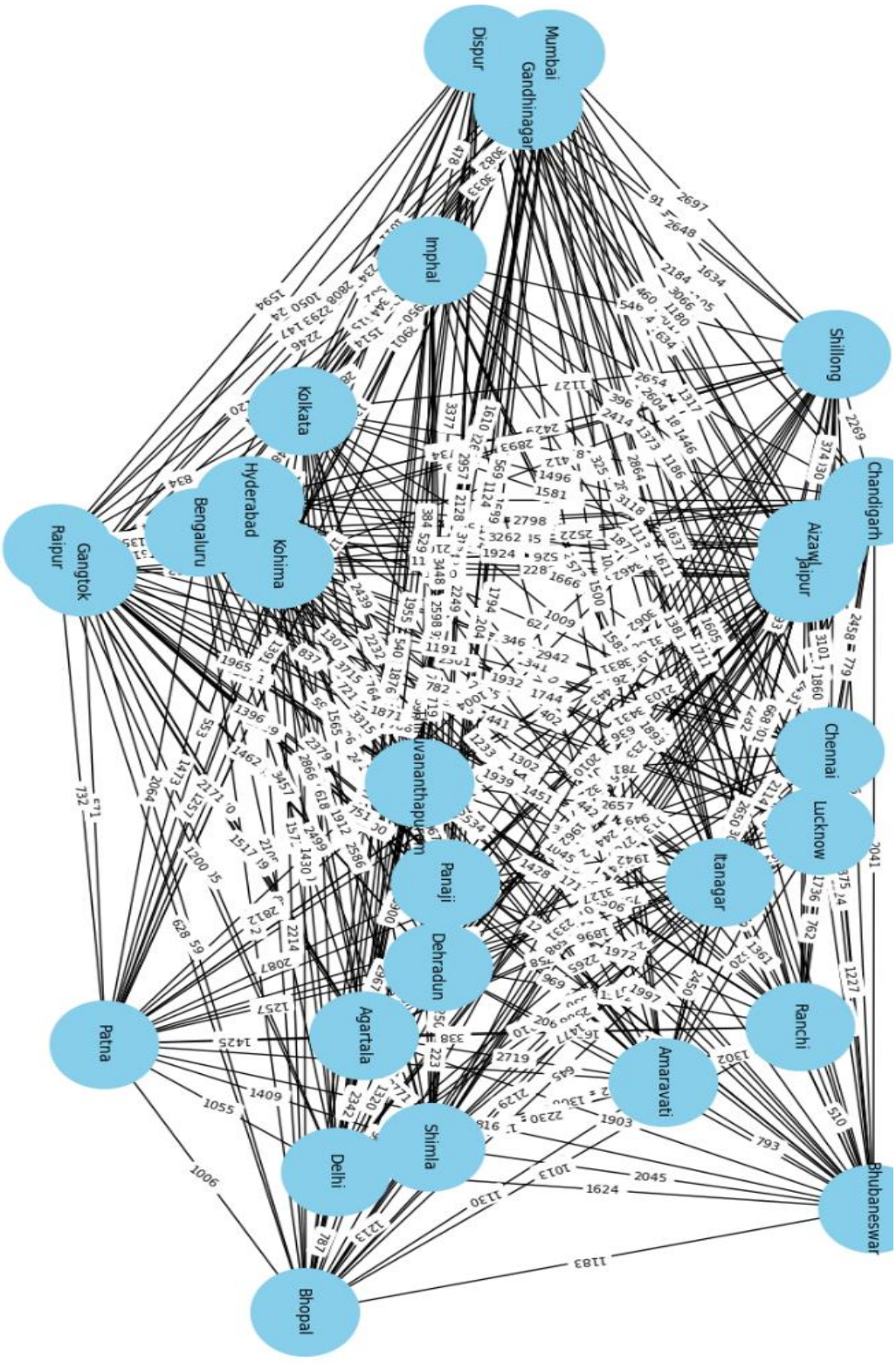


Figure 10: Graph generated for the TSP considering all capitals of India

10C. Source code of the program used

algo comapre.py | Release Notes: 1.85.2

C: > Users > poshi > OneDrive > Desktop > CS EE > algo comapre.py > ...

```python
import networkx as nx
import time

# Define the distances as nested dictionaries
distances = {
    "Delhi": {
        "Amaravati": 1204,
        "Chandigarh": 244,
        "Dehradun": 245,
        "Gandhinagar": 970,
        "Gangtok": 1605,
        "Hyderabad": 1573,
        "Itanagar": 2210,
        "Jaipur": 306,
    },
    "Amaravati": {
        "Chandigarh": 2114,
        "Dehradun": 2066,
        "Gandhinagar": 1500,
        "Gangtok": 1900,
        "Hyderabad": 280,
        "Itanagar": 2450,
        "Jaipur": 1730,
    },
    "Chandigarh": {
        "Dehradun": 171,
        "Gandhinagar": 1105,
        "Gangtok": 1858,
        "Hyderabad": 1832,
        "Itanagar": 2431,
        "Jaipur": 482,
    },
    "Dehradun": {
        "Gandhinagar": 1174,
        "Gangtok": 1545,
        "Hyderabad": 1786,
        "Itanagar": 2107,
        "Jaipur": 556,
    },
    "Gandhinagar": {
        "Gangtok": 2246,
        "Hyderabad": 1207,
        "Itanagar": 2805,
        "Jaipur": 634,
    },
    "Gangtok": {
        "Hyderabad": 2024,
        "Itanagar": 766,
    },
    "Hyderabad": {
        "Itanagar": 2585,
        "Jaipur": 1445,
    },
    "Itanagar": {
        "Jaipur": 2201,
    },
}

# Set positions for all nodes manually
pos = {
    "Delhi": (0, 0),
    "Amaravati": (1, 4),
    "Chandigarh": (3, 0),
    "Dehradun": (3, 2),
    "Gandhinagar": (2, 2),
    "Gangtok": (1, 1),
    "Hyderabad": (1, 3),
    "Itanagar": (2, 4),
    "Jaipur": (2, 0),
}

# Create the Hamiltonian graph with your specified dataset
G = nx.DiGraph()

for source, destinations in distances.items():
    for destination, distance in destinations.items():
        G.add_edge(source, destination, weight=distance)
```

```python
}

# Create the Hamiltonian graph with your specified dataset
G = nx.DiGraph()

for source, destinations in distances.items():
    for destination, distance in destinations.items():
        G.add_edge(source, destination, weight=distance)

# Measure runtime of Dijkstra's algorithm
start_time = time.time()
shortest_paths_dijkstra = nx.single_source_dijkstra_path_length(G, source='Delhi', weight='weight')
dijkstra_runtime = time.time() - start_time

# Measure runtime of Bellman-Ford algorithm
start_time = time.time()
shortest_paths_bellman_ford = nx.single_source_bellman_ford_path_length(G, source='Delhi', weight='weight')
bellman_ford_runtime = time.time() - start_time

# Compare the runtimes
print("Dijkstra's Runtime: {:.6f} seconds".format(dijkstra_runtime))
print("Bellman-Ford Runtime: {:.6f} seconds".format(bellman_ford_runtime))

# Check if the results are the same (both should give the same shortest paths)
if shortest_paths_dijkstra == shortest_paths_bellman_ford:
    print("Dijkstra and Bellman-Ford produce the same shortest paths.")
else:
    print("Dijkstra and Bellman-Ford produce different shortest paths.")
print("Shortest Paths (Dijkstra):")
for destination, path in shortest_paths_dijkstra.items():
    print(f"To {destination}: {path}")

# Print the shortest paths produced by Bellman-Ford algorithm
print("\nShortest Paths (Bellman-Ford):")
for destination, path in shortest_paths_bellman_ford.items():
    print(f"To {destination}: {path}")
```

```python
import networkx as nx
import matplotlib.pyplot as plt

# Create an empty graph
G = nx.Graph()

# Define the data as a dictionary
data = {
    "Delhi": {
        "Amaravati": 1204,
        "Chandigarh": 244,
        "Dehradun": 245,
        "Gandhinagar": 970,
        "Gangtok": 1605,
        "Hyderabad": 1573,
        "Itanagar": 2210,
        "Jaipur": 306,
        "Shimla": 343,
        "Ranchi": 1306,
        "Bengaluru": 2105,
        "Thiruvananthapuram": 2967,
        "Bhopal": 787,
        "Mumbai": 1416,
        "Imphal": 2424,
        "Shillong": 2010,
        "Aizawl": 2272,
        "Bhubaneswar": 1624,
        "Chennai": 2265,
        "Kohima": 2300,
        "Lucknow": 550,
        "Panaji": 1900,
        "Patna": 1055,
        "Agartala": 2342,
        "Kolkata": 1472,
        "Raipur": 1200,
        "Dispur": 1955,
    }
}
```

```
        "Dispur": 2957,
    },
    "Patna": {
        "Agartala": 1425,
        "Kolkata": 553,
        "Raipur": 732,
        "Dispur": 903,
    },
    "Agartala": {
        "Kolkata": 1565,
        "Raipur": 2171,
        "Dispur": 529,
    },
    "Kolkata": {
        "Raipur": 834,
        "Dispur": 1011,
    },
    "Raipur": {
        "Dispur": 1594,
    },
}

# Create a graph using NetworkX
G = nx.Graph()

# Add nodes (cities) to the graph
for city in distances:
    G.add_node(city)

# Add edges (distances between cities) to the graph
for city, city_distances in distances.items():
    for destination, distance in city_distances.items():
        G.add_edge(city, destination, weight=distance)

# Create a Hamiltonian map with a different layout (e.g., spring_layout)
pos = nx.spring_layout(G, seed=42)
```

```

# Increase the figure size for a clearer view
plt.figure(figsize=(20, 20))

# Draw the graph
nx.draw(G, pos, with_labels=True, node_size=4000, node_color='skyblue', font_size=10)
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels, font_size=8)

# Calculate the shortest paths using Dijkstra's algorithm and measure runtime
start_time = time.time()
dijkstra_shortest_paths = dict(nx.all_pairs_dijkstra_path(G))
dijkstra_runtime = time.time() - start_time

# Calculate the shortest paths using Bellman-Ford algorithm and measure runtime
start_time = time.time()
bellman_ford_shortest_paths = dict(nx.all_pairs_bellman_ford_path(G))
bellman_ford_runtime = time.time() - start_time

# Compare and print the shortest paths
print("Dijkstra's Runtime:", dijkstra_runtime, "seconds")
print("Bellman-Ford Runtime:", bellman_ford_runtime, "seconds")

# Example: Print shortest path from Delhi to Chennai
source = "Delhi"
destination = "Chennai"
shortest_path_dijkstra = dijkstra_shortest_paths[source][destination]
shortest_path_bellman_ford = bellman_ford_shortest_paths[source][destination]
print(f"Shortest Path from {source} to {destination} (Dijkstra):", shortest_path_dijkstra)
print(f"Shortest Path from {source} to {destination} (Bellman-Ford):", shortest_path_bellman_ford)

# Show the Hamiltonian map
plt.show()
```

```python
import networkx as nx

cities = ['Amaravati', 'Chandigarh', 'Dehradun', 'Gandhinagar', 'Gangtok', 'Hyderabad', 'Itanagar', 'Jaipur', 'Shimla', 'Ranchi',
          'Bengaluru', 'Thiruvananthapuram', 'Bhopal', 'Mumbai', 'Imphal', 'Shillong', 'Aizawl', 'Bhubaneswar', 'Chennai',
          'Kohima', 'Lucknow', 'Panaji', 'Patna', 'Agartala', 'Kolkata', 'Raipur', 'Dispur']

distances = {('Amaravati', 'Delhi'): 1204, ('Chandigarh', 'Delhi'): 244, ('Dehradun', 'Delhi'): 245, ('Gandhinagar', 'Delhi'): 970,
             ('Gangtok', 'Delhi'): 1605, ('Hyderabad', 'Delhi'): 1573, ('Itanagar', 'Delhi'): 2210, ('Jaipur', 'Delhi'): 306,
             ('Shimla', 'Delhi'): 343, ('Ranchi', 'Delhi'): 1306, ('Bengaluru', 'Delhi'): 2105, ('Thiruvananthapuram', 'Delhi'): 2967,
             ('Bhopal', 'Delhi'): 787, ('Mumbai', 'Delhi'): 1416, ('Imphal', 'Delhi'): 2424, ('Shillong', 'Delhi'): 2010,
             ('Aizawl', 'Delhi'): 2272, ('Bhubaneswar', 'Delhi'): 1624, ('Chennai', 'Delhi'): 2265, ('Kohima', 'Delhi'): 2300,
             ('Lucknow', 'Delhi'): 550, ('Panaji', 'Delhi'): 1900, ('Patna', 'Delhi'): 1055, ('Agartala', 'Delhi'): 2342,
             ('Kolkata', 'Delhi'): 1472, ('Raipur', 'Delhi'): 1200, ('Dispur', 'Delhi'): 1955}

graph = nx.Graph()

for city in cities:
    graph.add_node(city)

for (city1, city2) in distances:
    graph.add_edge(city1, city2, weight=distances[city1, city2])

print(graph)
```

```python
import networkx as nx
import matplotlib.pyplot as plt

# Import the pyTSP library
import pyTSP

# Create a list of cities
cities = ['Amaravati', 'Chandigarh', 'Dehradun', 'Gandhinagar', 'Gangtok', 'Hyderabad', 'Itanagar', 'Jaipur', 'Shimla', 'Ranchi',
          'Bengaluru', 'Thiruvananthapuram', 'Bhopal', 'Mumbai', 'Imphal', 'Shillong', 'Aizawl', 'Bhubaneswar', 'Chennai',
          'Kohima', 'Lucknow', 'Panaji', 'Patna', 'Agartala', 'Kolkata', 'Raipur', 'Dispur']

# Create a dictionary of distances between cities
distances = {('Amaravati', 'Delhi'): 1204, ('Chandigarh', 'Delhi'): 244, ('Dehradun', 'Delhi'): 245, ('Gandhinagar', 'Delhi'): 970,
             ('Gangtok', 'Delhi'): 1605, ('Hyderabad', 'Delhi'): 1573, ('Itanagar', 'Delhi'): 2210, ('Jaipur', 'Delhi'): 306,
             ('Shimla', 'Delhi'): 343, ('Ranchi', 'Delhi'): 1306, ('Bengaluru', 'Delhi'): 2105, ('Thiruvananthapuram', 'Delhi'): 2967,
             ('Bhopal', 'Delhi'): 787, ('Mumbai', 'Delhi'): 1416, ('Imphal', 'Delhi'): 2424, ('Shillong', 'Delhi'): 2010,
             ('Aizawl', 'Delhi'): 2272, ('Bhubaneswar', 'Delhi'): 1624, ('Chennai', 'Delhi'): 2265, ('Kohima', 'Delhi'): 2300,
             ('Lucknow', 'Delhi'): 550, ('Panaji', 'Delhi'): 1900, ('Patna', 'Delhi'): 1055, ('Agartala', 'Delhi'): 2342,
             ('Kolkata', 'Delhi'): 1472, ('Raipur', 'Delhi'): 1200, ('Dispur', 'Delhi'): 1955}

# Create a graph
graph = nx.Graph()

# Add nodes to the graph
for city in cities:
    graph.add_node(city)

# Add edges to the graph
for (city1, city2) in distances:
    graph.add_edge(city1, city2, weight=distances[city1, city2])

# Find a Hamiltonian path in the graph
path = pyTSP.solve_tsp(graph)

# Draw the graph using Matplotlib
plt.figure()
nx.draw(graph, with_labels=True, path=path)
plt.show()
```

```python
import networkx as nx
import time

import matplotlib.pyplot as plt

# Create an empty graph
G = nx.Graph()

# Define the data as a dictionary
data = {
    "Delhi": {
        "Amaravati": 1204,
        "Chandigarh": 244,
        "Dehradun": 245,
        "Gandhinagar": 970,
        "Gangtok": 1605,
        "Hyderabad": 1573,

    }
}

# Add nodes and edges to the graph based on the data
for source, destinations in data.items():
    for destination, distance in destinations.items():
        G.add_edge(source, destination, weight=distance)

# Visualize the graph
pos = nx.spring_layout(G)
labels = nx.get_edge_attributes(G, 'weight')
nx.draw(G, pos, with_labels=True, node_size=500, node_color="skyblue", font_size=12)
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.title("Hamiltonian Map of Capital Cities")
plt.show()

```

```python
plt.title("Hamiltonian Map of Capital Cities")
plt.show()


# Select the source city (e.g., "Delhi")
source_city = "Delhi"

# Measure the runtime of the Bellman-Ford algorithm
start_time = time.time()

# Calculate the shortest paths from the source city to all other cities
shortest_paths = nx.single_source_bellman_ford(G, source_city)

end_time = time.time()

# Calculate the runtime
runtime = end_time - start_time

# Print the shortest paths and runtime
for city, path_info in shortest_paths.items():
    if city != source_city:
        shortest_path, shortest_distance = path_info
        print(f"Shortest path from {source_city} to {city}: {shortest_path}, Distance: {shortest_distance} km")

print("Bellman-Ford runtime:", runtime, "seconds")

```