JOURNAL OF
EMERGING INVESTIGATORS

Article

Comparing traveling salesman problem algorithms to chart delivery routes in a city

Aarav Jalan[1], Pedro D. Bello-Maldonado[2]
[1] Dhirubhai Ambani International School, Mumbai, Maharashtra, India
[2] Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, Illinois

SUMMARY

Delivery and logistics companies such as Swiggy, Uber Eats, and Zomato strive to deliver to customers as fast as possible by traveling the shortest, most optimal routes to reduce travel time. The travelling salesman problem (TSP) is a combinatorial optimization problem that attempts to minimize tour lengths when traversing edges in a mathematical graph. Given that the TSP is a computationally heavy problem, modern computers lack the processing power needed to find an exact, optimal solution, which requires large amounts of recursion and backtracking. In a city, nodes represent households or neighborhoods, while edges represent the roads traveled by delivery vans. Therefore, the TSP offers a solution to reduce travel time in cities by minimizing the distances traveled on roads. Our study compared different TSP algorithms and heuristics used to estimate solutions. We implemented multiple approximation algorithms and determined the accuracy and efficiency of these models using real-world datasets. Further, we hypothesized that increasing the number of delivery destinations will result in higher total travel times, and that the choice of logistics algorithm will significantly influence both the estimated travel distance and the algorithmic compute time when charting optimal travel routes. Our results indicated that with more delivery destinations, longer routes produced greater travel times. Additionally, we saw that simpler TSP algorithms estimated longer tour lengths, but also took shorter runtimes to compute the solution than more complex TSP algorithms. Finally, our findings gave us insights into the potential of utilizing the TSP to chart delivery routes in a city.

INTRODUCTION

Optimization problems have become increasingly prevalent in delivery services today, with new algorithms being developed to reduce computational power and increase the efficiency of travel route results. In the context of travel routes, optimizing the path taken to reach from one point to another is very important for delivery companies as it allows them to reduce travel time. A well-known optimization problem in planning and delivery logistics is the traveling salesman problem (TSP): a combinatorial optimization problem that attempts to minimize tour lengths when traversing edges in a mathematical graph. While in a realistic scenario, several other factors such as traffic congestion, wind speed, and fuel efficiency might affect travel time, minimizing the distance traveled by road is one of the most important variables that is controlled by the driver who chooses the route (1).

Prior research on travel route optimization has explored a wide range of approaches to solving the TSP. A notable example is the work of team "Just Passing Through" in Amazon's 2021 Last Mile Routing Research Challenge, which demonstrated how artificial intelligence (AI) can be combined with the TSP to create efficient delivery sequences based on real-world datasets and constraints (2). This challenge not only focused on the minimization of travel distances but also determined tours that closely match those taken by experienced drivers. Similarly, other researchers have investigated neural network methods, viewing the TSP as a weighted graph problem and optimizing routes through objective functions (3). These works provided us with insights into how machine learning can complement traditional optimization techniques in adapting to complex data and city layouts. In addition to AI-based approaches, foundational algorithmic studies have contributed significantly to understanding the TSP through various heuristics and algorithms, such as nearest neighbor, insertion methods, and Christofides, highlighting their computational complexity and performance on different kinds of graphs (4). These works helped us select our algorithms by showing the practical trade-offs between computational speed and route accuracy.

The TSP can be used to find the shortest distance when traveling between multiple houses in a neighborhood: in a mathematical graph, nodes correspond to houses, while edges represent paths or roads between them. While actual cities involve curved roads, traffic patterns, and one-way streets, graphs simplify this by modeling roads as straight, two-way edges and assuming uniform conditions for computational feasibility. An optimal path is the shortest and fastest route between two or more nodes in a graph. A suboptimal path is not the shortest route but one that still succeeds in connecting the desired nodes while being less efficient in comparison. The TSP is classified as a nondeterministic polynomial-time hard (NP-hard) problem. This means that no graph-search algorithm can determine the exact, optimal path to the TSP in polynomial time, which is time that scales with input size as a power of n, like n^2, n^3, etc. In practice, suboptimal paths are found using heuristic algorithms and can still provide viable, approximated solutions (5).

The TSP is defined as symmetric on an undirected graph (paths have no specific direction), and asymmetric on a directed one (only one direction, like one-way roads). Graphs can be sparse (not all nodes are interconnected) or connected. In this study, we considered the symmetric TSP and connected graphs, where each node has edges connecting it to other nodes.

We explored five TSP optimization algorithms: nearest neighbor, nearest insertion, cheapest insertion, convex hull

https://doi.org/10.59720/24-210

insertion, and Christofides. All edges were considered to be undirected, as this condition is required by the TSP algorithms to ensure every node is visited.

Nearest neighbor approximation algorithms adopt a greedy approach and construct tours by repeatedly visiting the nearest unvisited city (6). The algorithm starts at an arbitrary city, and always chooses the nearest, unvisited node with the smallest travel cost (given by the Cartesian distance formula). After moving to the nearest unvisited city, it is marked as visited, and this is repeated until all nodes are visited once, after which the algorithm returns to the starting node.

Nearest insertion algorithms build tours by starting with a small route (usually two cities), and then repeatedly finding the unvisited city that's closest to any city in the current route (7). Instead of just adding this new city to the end, the algorithm tries inserting it between each pair of cities in the route and chooses the spot where it increases the total travel distance the least. This makes it more flexible than nearest neighbor, which always extends the route from the current city.

Cheapest insertion algorithms insert a node at the position that minimizes the cost at each step, irrespective of nearby proximity (8). It starts with a partial tour of two cities, and then for every unvisited city, calculates the cost of inserting a new node k where there is most minimization on the cost between every pair of existing nodes in the tour (i and j). It aims to minimize the net cost of traveling from i to k to j. The node k must minimize: $c_x = c_{ik} + c_{kj} – c_{ij}$ where c_{ij} is the travel cost from i to j. This relates to the triangle inequality: $c_{ik} + c_{kj} \geq c_{ij}$. The greatest minimization is when cost $c_{ik} + c_{kj} = c_{ij} \rightarrow c_x = 0$, as there is no cost increase on the addition of a new node. While nearest insertion chooses the closest unvisited node, then finds the best place to insert it, cheapest insertion finds the node *and* position that causes the smallest increase in total cost (8).

Convex hull insertion starts by connecting the outermost cities (those that form the boundary of the graph) into a loop called a convex hull (9). Then, it adds the remaining cities (inside the loop) at the place where they increase the total travel distance the least. It uses polygons to mathematically fit the data points, and the combination of building a basic outer path first and then inserting the rest of the cities strategically helps make the overall route more efficient (9).

Christofides algorithms find solutions within 1.5 times the optimal tour length for the metric TSP. The algorithm first constructs a minimum spanning tree (MST), a subgraph that connects all cities with the smallest possible total edge weight. Next, it finds odd-degree vertices in the MST, and pairs these together using an algorithm. This adds additional edges to the MST to make all vertex degrees even. Finally, all of these edges from the MST are combined with a Eulerian circuit to form a Hamiltonian cycle, a graph cycle that visits each node exactly once (10). The use of several optimization techniques makes this the most complex algorithm explored in this paper.

An algorithm's complexity is a function that defines its efficiency based on the amount of data it must process. It's taken as a function of n, the number of nodes, and provides a mathematical estimation of the runtime required by the algorithm to calculate the solution. Additionally, the computational cost of an algorithm, or resources required to run it, is proportional to its complexity.

The complexities of both the nearest neighbor and nearest insertion are $O(n^2)$, where O is a function of n and refers to the worst-case complexity analysis (7). Cheapest insertion and convex hull have complexities of $O(n^2 \log_2(n))$, and Christofides has a complexity of $O(n^3)$ (4). Practically, this suggests that nearest neighbor and nearest insertion require the least computational time to determine a solution, while Christofides is the slowest but often the most accurate.

In this study, we compared the effectiveness of these different graph-search algorithms by testing them on common synthetic data. We hypothesized that an increase in the number of delivery destinations would result in greater total travel times, and that the choice of TSP algorithm (among the five we tested) would significantly influence both the estimated travel distance and the algorithmic compute time when charting optimal travel routes. Our hypothesis was supported by our data because with more nodes in a graph, the estimated travel distance increased.

Also, while testing our hypothesis, the Christofides algorithm, given its highest complexity, had the most accurate estimation of the optimal tour length found in the literature, while the nearest neighbor was least accurate, given its smallest complexity (11–13). The nearest neighbor algorithm had the smallest runtime, given its low complexity. These results helped us evaluate when a particular TSP algorithm should be used given time or accuracy constraints.

RESULTS

In this study, we investigated the effectiveness of different approximation algorithms when tested on two synthetic datasets that modeled real-world Euclidean distances. These datasets were extracted from the TSPLIB library, which was compiled by Gerhard Reinelt, and taken from Rutgers's Center for Discrete Mathematics and Theoretical Computer Science's (DIMACS) 8th Implementation Challenge (11, 12). These datasets, called C1k.1 and E1k.1, had different city layouts. They contained 1,000 nodes each with optimal tour lengths of 11,376,735 and 22,985,695 arbitrary units, respectively, as given on the source website.

We created a Python program to compare two TSP algorithms: nearest neighbor and cheapest insertion (13). We built a TSP solver application to run our program more easily (**Figure 1**). We tested the efficiency and accuracy of these algorithms on the above datasets, obtaining tour length estimations. However, the tour length data on these datasets for the other algorithms (cheapest insertion, convex-hull, and Christofides) were taken from the DIMACS reference website, as we didn't program our algorithms for these (12). We tested two main parameters: the estimated tour length and the runtime taken to determine this, as they reflected a key trade-off in route optimization: speed versus solution quality. Faster algorithms may run quickly but often produce longer routes, increasing travel time and fuel use.

The percentage errors represent the relative variance of the estimated to the actual, optimal tour length (**Table 1**). Thus, smaller errors mean higher accuracy. On average, sample E1k.1, with a larger actual tour length, had fewer errors in its approximations. The results across both data samples were consistent, with the Christofides algorithm having least errors of 9.09% and 6.76%, respectively, making it the most accurate

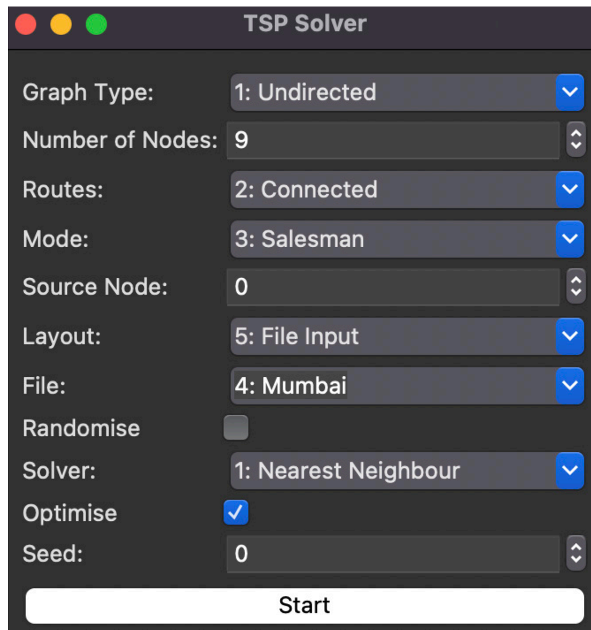


Figure 1: The TSP solver application. This screenshot depicts the features of the TSP solver developed for this study. It generated networkx graphs and estimated travel lengths, which were analyzed in this study. The output graph of this configuration is displayed in **Figure 4**.

approximation algorithm under these datasets of 1,000 cities (**Table 1**). It was followed by all three insertion algorithms, with the nearest insertion and cheapest insertion being quite close to another in their percentage errors due to their similar utilization of insertion heuristics. The nearest neighbor algorithm had the largest error among all the algorithms, making it the least efficient when estimating the shortest path. When comparing the two algorithms we developed for this study, we saw that the cheapest insertion algorithm was more accurate with 1,000 nodes, with lower errors of 19.63% and 19.13% compared to the nearest neighbor algorithm with higher errors of 24.52% and 22.87%, respectively (**Table 1**).

Additionally, specific nodes from sample E1k.1 were selected to compare the optimality and runtime of both algorithms with different numbers of nodes. When there were few nodes in the graph, the nearest neighbor and cheapest insertion algorithms calculated similar tour lengths, with the nearest neighbor sometimes being more accurate than the cheapest insertion algorithm (**Table 2**). But as the number of nodes increased, the cheapest insertion algorithm became more accurate, as observed with 100, 200, and 1,000 nodes (**Table 2**). Furthermore, as the number of nodes or the complexity of the graph increased, the difference between the optimal tour lengths by both algorithms for the same number of nodes also increased, as the nearest neighbor algorithm becomes less accurate with more nodes (**Figure 2**).

TSP Algorithm	Travel Distances (units)			
	C1k.1	% Error	E1k.1	% Error
Christofides*	12410651	9.09	24539508	6.76
Convex Hull Insertion*	13045792	14.67	26629300	15.85
Cheapest Insertion	13610371	19.63	27383697	19.13
Nearest Insertion*	13947570	22.60	28180189	22.60
Nearest Neighbor	14166622	24.52	28242959	22.87

Table 1: Tour lengths estimated by different TSP algorithms. The travel path distances (arbitrary units), and percentage errors generated by different TSP algorithms for 1,000 nodes. Results labeled with a * were taken from Rutgers's DIMACS 8th Implementation Challenge (12).

Tour improvement algorithms like 2-opt and 3-opt iteratively remove and reconnect edges to eliminate path crossings and reduce overall distance (**Figures 3A–B**). They further decreased the tour length for both algorithms (**Figure 3**). However, in certain cases, these tour improvement algorithms had no optimization effect, especially with fewer nodes (as with 10 and 25 nodes).

With fewer nodes, the algorithmic runtimes to calculate the tour distance appeared to be similar at 0.3 ms at 10 nodes and 0.4 ms and 0.5 ms at 20 nodes (**Table 2**). However, as the number of nodes in the graph increased, the cheapest insertion algorithm required a higher runtime. From identical runtimes at 10 nodes, with 200 nodes, the cheapest insertion algorithm (142.7 ms) took 140.1 ms more; thus the nearest neighbor algorithm had a shorter runtime (2.6 ms).

To test the real-world applicability of the TSP, we generated a networkx graph that charted out a route through the various neighborhoods (nodes) in the city of Mumbai (**Figure 4**). This route started and ended at the neighborhood Prabhadevi and used the nearest neighbor algorithm to determine the travel path. The locations of the nodes were actual geographical coordinates of the neighborhoods in India, while edges represented the roads from one sector to another, with the road lengths as edge weights. Plotting the path from the model graph on an actual map of Mumbai, our results showed quite a different route, as the roads were no longer linear and may not necessarily be bidirectional (**Figures 4, 5**). In fact, the distance of the new path was 70.1 km, or 25.1 km more than, or 55.78% longer than the estimated tour length of 45.0 km (**Figure 4**).

The differences between the estimated and actual tour lengths are almost all greater than 30% (**Table 3**). The theoretical models strongly deviate from the values of the practical real-world data. The biggest percentage difference of 49.31% from Mahalaxmi to Churchgate is because of the large non-linear curvature in the path between the two neighborhoods (**Figure 5**).

DISCUSSION

The purpose of our study was to determine whether more nodes (delivery destinations) resulted in greater travel times,

Nodes	Nearest Neighbor				Cheapest Insertion			
	Raw Run	2-opt	3-opt	Time (ms)	Raw Run	2-opt	3-opt	Time (ms)
10	3004029	3004029	2645154	0.3	3449066	2667172	2667172	0.3
25	3914140	3754171	3754171	0.4	4077915	3644511	3644511	0.5
50	7000327	6532948	6093307	0.6	5911612	5734176	5734176	12.5
100	9225559	8404151	7925558	1.5	8737276	8302285	8006174	19.1
200	13277675	12467335	12016354	2.6	12368675	11734251	11503131	142.7

Table 2: Tour lengths and runtimes of nearest neighbor and cheapest insertion algorithms. The tour lengths (arbitrary units) and runtimes of the nearest neighbor and cheapest insertion algorithms with different numbers of nodes for dataset E1k.1 were compared. 2-opt and 3-opt represent tour improvement algorithms. Time refers to the runtime taken by the raw-run algorithms to estimate the tour length, in milliseconds, without 2-opt and 3-opt.

https://doi.org/10.59720/24-210

Source	Destination	Tour Length (km)		% Difference
		Estimated	Actual	
Prabhadevi	Matunga	2.83	3.91	**27.62**
Matunga	Sion	1.38	2.57	**46.30**
Sion	Bandra	3.83	6.55	**41.53**
Bandra	Santacruz	2.59	3.85	**32.73**
Santacruz	Juhu	2.36	3.41	**30.79**
Juhu	Andheri	4.39	6.92	**36.56**
Andheri	Mahalaxmi	13.71	19.90	**31.11**
Mahalaxmi	Churchgate	5.49	10.83	**49.31**
Churchgate	Prabhadevi	8.42	12.16	**30.76**
Total		45.00	70.10	**35.81**

Table 3: Actual and estimated tour lengths of routes in Mumbai. Real-world tour lengths are compared to those generated by our algorithms to determine their effectiveness. The percentage difference was also calculated with reference to the actual length. This gives us an idea of how much our estimations deviate from the actual path lengths.

and whether the choice of TSP algorithm affected the travel distance and algorithmic runtime. We also aimed to test the real-world applicability of the TSP.

Our study demonstrated that Christofides algorithm was the most accurate algorithm for these datasets with 1,000 nodes. Further, the results between nearest neighbor and cheapest insertion showed that with few nodes, the algorithms work similarly in finding a nearly identical estimated tour distance. In certain cases, the nearest neighbor algorithm provided a more accurate, cost-effective solution. This is because the nearest neighbor algorithm, being a greedy algorithm, focuses on locally optimal solutions at each node, disregarding its effects on the future result (14). With few nodes, this was highly effective, as there were few edges to consider, and thus selecting the closest path may often result in a better solution. Heuristically, the nearest neighbor algorithm uses a single loop that determines the closest, unvisited, adjacent edge to the running path. However, this approach is often based on chance and depends on the nature of the edges in the graph: with few edges, there is less room for error, making this more accurate with fewer nodes. The cheapest insertion algorithm uses complex mathematical logic for approximation, but with few nodes, the effects of this are less seen. As the number of nodes increases, if the salesman continuously greedily picks the closest edge, it may result in a highly suboptimal solution.

The cheapest insertion algorithm's mathematical minimization of the cost increase between two nodes by adding a third in between helped logically improve the tour route (14). While the cheapest insertion algorithm considered the cheapest cost from any two nodes, the nearest neighbor algorithm only focused on one particular node's subtours at a point in time, making it less accurate with more nodes. This explained the observed differences in runtimes (**Table 2**).

The nearest neighbor algorithm has a time complexity of about n^2, while cheapest insertion takes slightly longer at around $n^2 \log_2(n)$, where n is a positive integer (15). When the number of nodes (n) is small, the extra $\log_2(n)$ factor doesn't make much difference, so both algorithms have a similar runtime. But as n increases, the $\log_2(n)$ factor becomes more noticeable, and cheapest insertion takes longer to find a solution, giving a larger difference in runtimes (as with our results). The plausible reason for this could lie in their approaches: the simple action of finding a minimum in the nearest neighbor algorithm might be less time-consuming than the several iterations, additions, and conditional statements in the cheapest insertion algorithm. It also must account for several iterations that occur when searching for and placing a new node between *any* two nodes. Conversely, the nearest neighbor algorithm must only deal with the neighbors of a single node, at a point in time, resulting in only one iteration per node.

The Christofides algorithm has the largest complexity of $O(n^3)$ compared to all the other algorithms, as $O(n^3) > O(n^2 \log_2(n)) > O(n^2)$, for more than 2 nodes ($n > 2$). Therefore, it also has the longest runtime and computational cost, as it involves the most nested iterations when determining the solution (16). These three results indicated a correlation between optimality and runtime: the compromise for higher accuracy and more optimal tour paths is that more time is required in calculations, hence a balance must be made between the two. This suggests that the complexity and runtime to achieve 100% accuracy might be too high. However, in certain cases, like when nearest nodes are also part of the optimal path in clustered or evenly spaced graphs, the nearest neighbor is both more accurate, based on the nature of the initial graph state.

Tour improvement algorithms 2-opt and 3-opt were applied to existing suboptimal routes generated by the other algorithms (17). These used recursive techniques to achieve greater accuracy, reducing the travel cost even further. A 2-opt algorithm removes two intersecting edges in a graph and reconnects them to prevent overlap, while 3-opt rearranges three intersecting edges, increasing accuracy by reducing the overall tour path length. The new length of all the colored edges, in both 2-opt and 3-opt, is shorter, thus the Euclidean distance traveled, and new tour length obtained are lesser, making the tour path shorter and more optimal (**Figures 3A–B)**.

2-opt and 3-opt helped in improving the travel route and were most effective when there were many nodes, as there were a lot more permutations of optimal paths available for the algorithm to consider. With just 10 and 25 nodes, both algorithms had little effect (raw run, 2-opt, and 3-opt were the

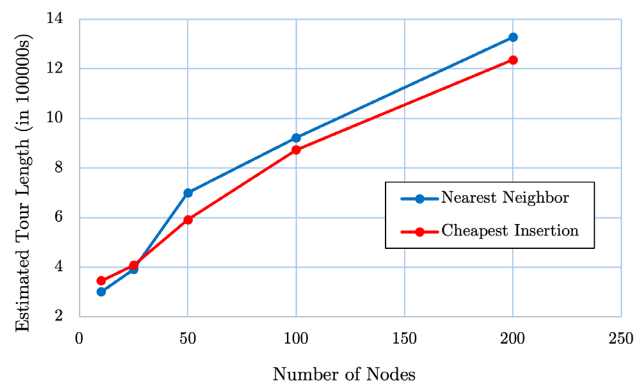


Figure 2: Tour length vs number of nodes. The number of nodes plotted against estimated tour lengths of the algorithms by incorporating the data from **Table 1**. Nearest neighbor has greater increases in tour lengths with more nodes.

https://doi.org/10.59720/24-210

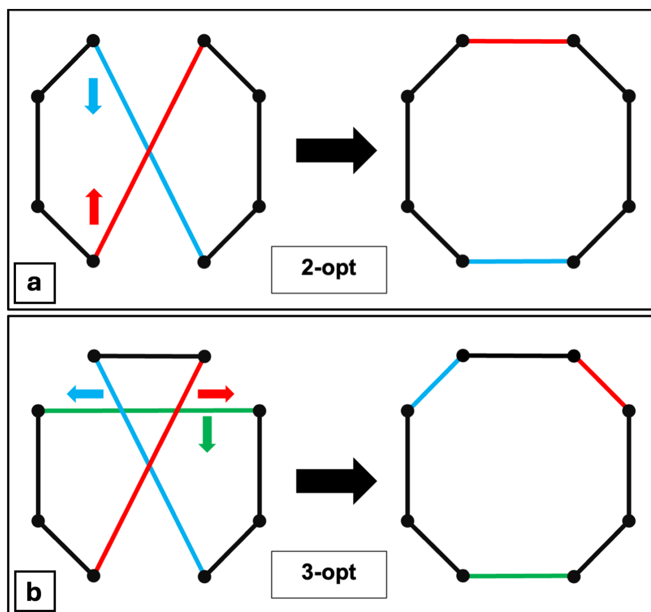


Figure 3: Representation of tour improvement algorithms. The (a) 2-opt and (b) 3-opt tour improvement algorithms remove two and three intersecting edges, respectively, and rearrange them to form more optimal and shorter routes.

same), as the nearest neighbor and cheapest insertion would often compute the most accurate path in the first attempt, given that there were few possibilities.

During this study, 2-opt and 3-opt algorithms were also applied to both datasets with 1,000 nodes, but due to the high amounts of recursion utilized by these algorithms, the runtime was too high to determine a solution (the runs never completed), hence the increased processing time of these tour improvement algorithms is a downside. This inability to find a highly accurate solution reiterates how the TSP cannot be feasibly solved in polynomial time but instead approximated.

In the real world, the TSP algorithms used for charting travel routes are much more complex, primarily because there are thousands of nodes (destinations) and edges (roads) traveled. Moreover, some factors cannot easily be modeled: roads are not straight paths like edges. Certain roads are unidirectional because of an asymmetric TSP, and factors like traffic congestion complicate our results as the shortest distance may not always be the fastest route (18).

Thus, the TSP algorithms explored in this paper cannot provide an exact value for the most accurate path that must be traveled to minimize travel time. However, they can be used for approximations and estimations, accounting for certain assumptions: all distances and paths are Euclidean (straight lines), there is no traffic, and all roads are bidirectional (two-way). Our models failed to account for the non-linearity of real-world roads, leading to an inaccuracy in the estimated travel distance in Mumbai when compared to the actual tour length (**Figures 4, 5**).

However, since the Euclidean distance from one neighbor to another indicates their proximity, our TSP algorithms might efficiently determine a correct path but not the correct cost, as the linear distance between points is often proportional to their actual distance (subject to general road layouts). Thus, the path obtained may just be the optimal (or approximately optimal) path for the map of Mumbai, but not the exact tour distance or length (**Figures 4, 5**).

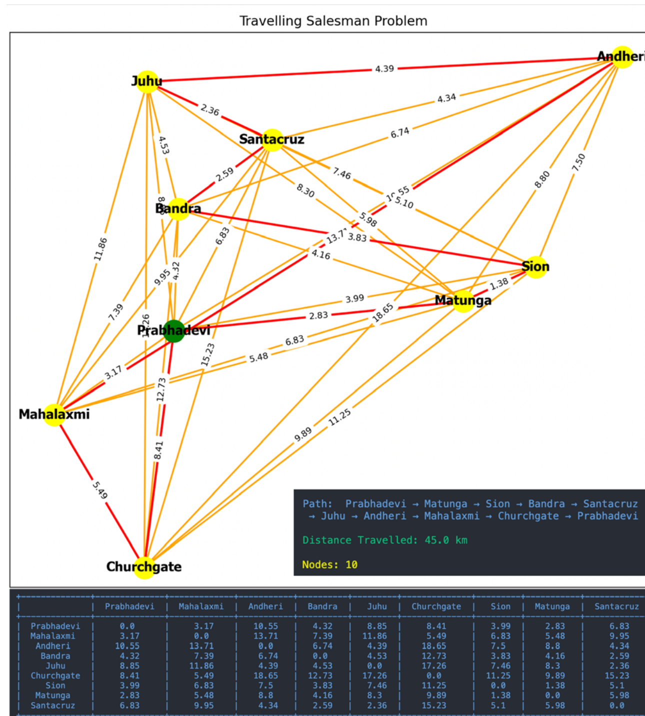
While determining which TSP algorithm is most applicable and efficient for use by delivery services, there are two factors to consider: how short and optimized the route is, and how much time is required to compute the route. In the real world, logistics services aim to reduce the total time taken, *T*, for a delivery process, which can be expressed as in **Equation 1**:

$$T = \text{Travel Time} + \text{Compute Time}$$

$$\text{Minimize } T = \frac{\text{Estimated Tour Length}}{\text{Speed of Vehicle}} + \text{Complexity of Algorithm} \quad \text{[Eqn 1]}$$

In theoretical applications, the TSP fails to account for the speed of the vehicle and only focuses on the tour length and algorithmic complexity. Tour length and complexity are inversely correlated, hence, there must be a reasonable trade-off between the two to achieve accuracy. Since v and w are unknown in a theoretical situation, we cannot simply choose one algorithm over another, as it depends on the scenario: if the vehicle travels slowly, the focus is on optimizing distance, and if fast, then the computation time should be limited as the travel time is lesser.

The nearest neighbor algorithm, with minimum iterations and conditional statements, will have the lowest runtime of $wO(n^2)$. However, its simpler mathematics and greedy approach of only focusing on a single node at a time also mean that c_{ij} will be more suboptimal. Contrarily, the cheapest insertion and nearest insertion algorithms are more forward-



	Prabhadevi	Mahalaxmi	Andheri	Bandra	Juhu	Churchgate	Sion	Matunga	Santacruz
Prabhadevi	0.0	3.17	10.55	4.32	8.85	8.41	3.99	2.83	6.83
Mahalaxmi	3.17	0.0	13.71	7.39	11.86	5.49	6.83	5.48	9.95
Andheri	10.55	13.71	0.0	6.74	4.39	18.65	7.5	8.8	4.34
Bandra	4.32	7.39	6.74	0.0	4.53	12.73	3.83	4.16	2.59
Juhu	8.85	11.86	4.39	4.53	0.0	17.26	7.46	8.3	2.36
Churchgate	8.41	5.49	18.65	12.73	17.26	0.0	11.25	9.89	15.23
Sion	3.99	6.83	7.5	3.83	7.46	11.25	0.0	1.38	5.1
Matunga	2.83	5.48	8.8	4.16	8.3	9.89	1.38	0.0	5.98
Santacruz	6.83	9.95	4.34	2.59	2.36	15.23	5.1	5.98	0.0

Figure 4: Theoretical travel routes in Mumbai. This networkx-generated mathematical graph displays a Euclidean model of the delivery routes in Mumbai, generated by the TSP Solver used in our research. The terminal output displays the adjacency or cost matrix, estimated tour length (45.0 km), and travel path in Mumbai when using the nearest neighbor algorithm.

https://doi.org/10.59720/24-210

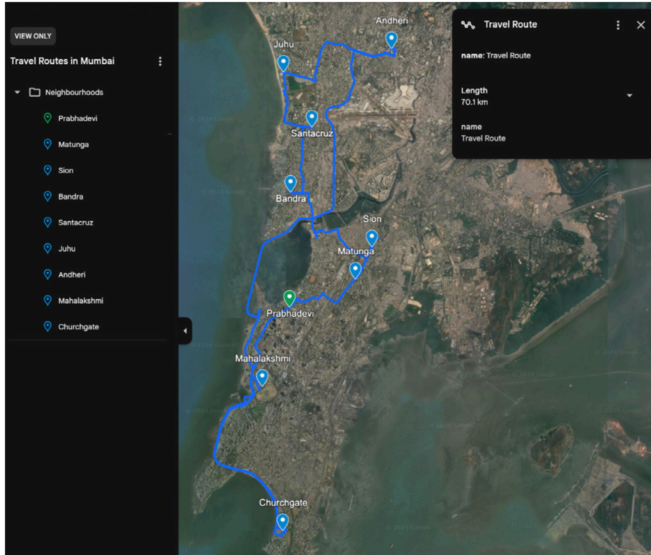


Figure 5: Real-world travel routes in Mumbai. A Google Earth map representing the real-world tour path and length (70.1 km) between neighborhoods in Mumbai, accounting for the non-linearity of roads and the fact that not all nodes are directly connected to each other.

thinking, using complex mathematics, iterative nested loops, and conditional statements that logically determine the largest benefit of minimizing the paths of existing locations. Since these algorithms focus on multiple nodes and the overall tour, c_{ij} will be lower and more accurate. That said, due to more iterations, its complexity of $O(n^2 \log_2(n))$ results in a greater compute time. The Christofides algorithm will result in an even smaller c_{ij} as it is the most accurate TSP solver in our study. However, this comes at the cost of a larger compute time of $wO(n^3)$. For shorter routes, the nearest neighbor algorithm might be most effective: as explored, it had similar travel paths to cheapest insertion, but much smaller runtimes. Therefore, it might be worth sacrificing a little more distance for lesser compute time in certain cases.

For all these algorithms, several calculations are required, which are carried out through mapping software such as Google Maps, Apple Maps, and even the TSP Solver developed for this study. Logically, the nearest neighbor algorithm can be followed manually, if a delivery person simply repeatedly chooses the closest destination to him and travels to it, without accounting for the overall route, making it intuitive for a traveling salesman.

That said, we must also consider the limitations and scalability of our algorithms in complex real-world scenarios. Nearest neighbor algorithms prioritize locally optimal solutions without considering their global impact, making them unreliable when traveling large road distances. Its greedy approach doesn't re-optimize previously chosen paths, causing inefficiencies to compound with each step, resulting in deviations from the optimal tour. Conversely, the insertion and Christofides algorithms have long compute times when producing solutions, making them impractical for large-scale, thousand-destination deliveries. For 2-opt and 3-opt algorithms, their recursive refinement methods, although helping improve suboptimal routes, are computationally difficult for dense graphs with many edges due to exponential growth in permutations they must evaluate. This occurred in our study, where the algorithms failed to produce 2-opt and 3-opt data within a feasible timeframe for datasets with 1,000 nodes (data not displayed: no results even after three hours with the tour-improvement algorithms, compared to a few seconds without).

Structural assumptions, like symmetrical costs and bidirectional connections between nodes, prevent TSP algorithms from accounting for real-world constraints like one-way roads, reducing their practical utility in urban delivery networks. The failure to incorporate directional road data can lead to non-traversable tours. For instance, the nearest neighbor selects the closest unvisited node, without considering the feasibility of the path under traffic regulations. When proceeding to the next node, it can encounter a situation where the direct path is inaccessible due to one-way restrictions, forcing the algorithm to backtrack or take longer detours to visit the remaining nodes. This can result in illegal or inefficient routes, increasing both the travel distance and time.

Addressing these challenges requires treating road networks as a directed graphs where edge weights represent asymmetric travel costs, and implementing legal restrictions and real-time traffic conditions. Asymmetric TSP solvers and ant colony optimization algorithms may help with such complexities by accounting for direction in their path generation processes (19).

MATERIALS AND METHODS

Creating the TSP solver application

We created a Python algorithm utilizing the networkx, NumPy, and matplotlib libraries to calculate and display weighted graph networks. Later, the algorithm included a tkinter application to determine traveling distances in graphs (**Figure 1**) (13). The application created both directed graphs (representing one-way roads) and undirected graphs (representing two-way roads). A variety of road layout options were available, including uploading preset files. Both graph-search algorithms were programmed in Python and tested in the application (13). Additional features included comparisons of algorithms, choices for optimization and randomization of paths, Euclidean graphs, and Dijkstra's greedy algorithm to find the path from one node to another (20).

General mathematics of the TSP

For the mathematics of the TSP, we began with a vertex set $V = \{1,2,3....,n\}$ consisting of a set of all nodes in a graph. Also, an edge set is represented as $E = \{ (i, j) \mid i, j \in V, i \neq j \}$. In connected graphs, each edge is bidirectional and an unordered pair of vertices: $\{ \{i, j\}, i < j \}$. Cost matrix $C = [c_{ij}]$, is a matrix of all the edge weights in the graph and has dimensions $n \times n$.

The TSP can be represented by and aims to minimize the following objective function (**Equation 2**):

$$\text{Minimize} \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \quad \forall \ i,j \in V \quad i \neq j \qquad \text{[Eqn 2]}$$

Here, c_{ij} represents the cost or Euclidean distance between nodes i and j, whereas x_{ij}, a binary parameter (either 0 or 1),

determines if an edge from i to j exists. We must account for constraints to ensure more optimality (21). Also, subtours are cycles that visit a subset of cities without covering all cities in a single tour. Subtour constraints minimize the number of subtours made, preventing unnecessary smaller cycles by ensuring that each node is visited only once:

$$\text{Entered Once: } \sum_{i=1,\ i\neq j}^{n} (x_{ij}) = 1\ \forall j \le n \quad \text{[Eqn 3]}$$

$$\text{Left Once: } \sum_{j=1,\ j\neq i}^{n} (x_{ij}) = 1\ \forall i \le n \quad \text{[Eqn 4]}$$

Equation 3 demonstrates that in the tour, there is only one edge entering node j from node i, for all possible j. For this edge, x_{ij} = 1, and for the remaining non-included edges x_{ij} = 0. Since only one value of x_{ij} is 1, and the rest are 0, the sum in **Equation 3**, which determines the number of edges going to every node j, is always 1. Similarly, **Equation 4** shows only one possible edge leaving node i to node j, for all nodes i. A Euclidean graph has multiple edges from i to any j. However, only one edge leaving i, for one particular value of j, is taken in the tour, where x_{ij} = 1. Therefore, the edge set for a tour path generated by a typical TSP algorithm can be represented as $E = \{ (i, j) \mid i, j \in V, i \neq j \}$, where one particular value of i and j appears no more than once.

Utilizing the TSP solver to optimize routes

The E1k.1 and C1k.1 synthetic datasets were entered into the TSP solver application multiple times to obtain metrics, including the tour length and runtime using both nearest neighbor and cheapest insertion algorithms (12). Tour improvements of 2-opt and 3-opt were enabled for up to 200 nodes. As explained, the runtime was too high for 1,000 nodes due to heavy processing: we obtained no results even after three hours, compared to few seconds at normal. A MacBook with an M1 Pro processor was used for this study, but with a more powerful CPU, the lower runtime may have made it possible to implement the tour improvement algorithms on 1,000 nodes.

Deeper analysis through benchmark data

Rutgers's "8th DIMACS Implementation Challenge" provided the synthetic data samples used in this study, which attempted to model real-world Euclidean city layouts (12). It also included benchmark results generated by other common TSP graph-search algorithms (apart from those of the study's TSP Solver), which were sample results obtained from computer programs of prior research. Instead of collecting all the primary data from a computer program, some secondary results from Rutgers' DIMACS website were used (12). These benchmark results were analyzed and included in this study to provide a holistic analysis of different TSP approaches (**Table 1**).

Comparison of TSP models and real-world paths

Google Earth's mapping software was used to construct a path and find the actual distance of the route when traveled by roads (**Figure 5**) (22). In the networkx graph, the Cartesian x and y coordinates of the nodes were the longitude and latitude locations of places on Earth (**Figure 4**). This way, the graph used real-world distances to model the delivery routes, resulting in fairer comparisons between the model and the actual path and distance.

Data analysis

The % error for each tour approximation was calculated using the following formula (**Equation 5**):

$$\%\ error = \left| \frac{\text{Optimal } (x) - \text{Approximated } (x)}{\text{Optimal } (x)} \right| \times 100 \quad \text{[Eqn 5]}$$

Here, x represents the tour length. Optimal (x) denotes the actual tour path length, while Approximated (x) is the estimated tour length generated by each TSP approximation algorithm.

Received: July 21, 2024
Accepted: February 10, 2025
Published: May 12, 2026

REFERENCES

1. D, Lakshmi. "What Is Traveling Salesman Problem and How Can Tech Solve It?" *Locus Blog*, 2 Mar. 2020, https://www.blog.locus.sh/travelling-salesman-problem-and-how-can-tech-solve-it/. Accessed 10 July 2024.
2. "Just Passing Through." *University of Waterloo*, https://www.math.uwaterloo.ca/tsp/amz/index.html. Accessed 25 March 2024.
3. Shi, Yong, and Yuanying Zhang. "The Neural Network Methods for Solving Traveling Salesman Problem." *Procedia Computer Science*, vol. 199, no. 1877-0509, 2022, pp. 681–686, https://doi.org/10.1016/j.procs.2022.01.084.
4. Nilsson, Christian. "Heuristics for the Traveling Salesman Problem." *Linkoping University*, vol. 38, no. 0085–9, Jan. 2003, https://pirun.ku.ac.th/~fengpppa/02206337/htsp.pdf. Accessed 26 March 2024.
5. "NP Hard Problems" *StudySmarter UK,* https://www.studysmarter.co.uk/explanations/computer-science/theory-of-computation/np-hard-problems/. Accessed 18 June 2024.
6. White, Leanne. "Solving the Traveling Salesman Problem." *RoadWarrior*, 21 Aug. 2023, https://roadwarrior.app/blog/solving-the-traveling-salesman-problem/. Accessed 26 June 2024.
7. Weru, Lawrence. "11 Animated Algorithms for the Traveling Salesman Problem." *STEM Lounge*, 28 Dec. 2019, https://www.stemlounge.com/animated-algorithms-for-the-traveling-salesman-problem/. Accessed 16 June 2024.
8. Fargiana, Farid, et al. "Implementation of Cheapest Insertion Heuristic Algorithm in Determining Shortest Delivery Route." *International Journal of Global Operations Research,* vol. 3, no. 2, 5 May 2022, pp. 37–45, https://doi.org/10.47194/ijgor.v3i2.137.
9. Kwon, Hyun, et al. "Algorithmic Efficiency in Convex Hull Computation: Insights from 2D and 3D Implementations." *Symmetry*, vol. 16, no. 12, 28 Nov. 2024, pp. 1590–1590, https://doi.org/10.3390/sym16121590.
10. Trung Luu, Quang. "Traveling Salesman Problem: Exact Solutions vs. Heuristic vs. Approximation Algorithms." *Baeldung*, 18 Mar. 2024, https://www.baeldung.com/

https://doi.org/10.59720/24-210

cs/tsp-exact-solutions-vs-heuristic-vs-approximation-algorithms. Accessed 17 June 2024.

11. Heidelberg University. "TSPLIB." *Comopt.ifi.uni-Heidelberg.de*, https://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Accessed 10 January 2024.
12. "TSP Challenge: Results Page." *Rutgers.edu*, 2019, https://archive.dimacs.rutgers.edu/Challenges/TSP/results.html. Accessed 10 January 2024.
13. Aarav Jalan. "Approximating Traveling Salesman Problem (TSP) Algorithms to Chart Delivery Routes in a City." 23 June 2024, https://github.com/AaravJalan/tsp-solver.
14. Rosenkrantz, Daniel J., et al. "An Analysis of Several Heuristics for the Traveling Salesman Problem." *SIAM Journal on Computing*, vol. 6, no. 3, Sept. 1977, pp. 563–581, https://doi.org/10.1137/0206041.
15. Cormen, Thomas H, et al. *Introduction to Algorithms*. Cambridge, Massachusetts, The Mit Press, 2022.
16. Christofides, Nicos. "Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem." *Apps.dtic.mil*, 1 Feb. 1976, https://apps.dtic.mil/sti/citations/ADA025602.
17. Lin, Shen. "Computer Solutions of the Traveling Salesman Problem." *Bell System Technical Journal*, vol. 44, no. 10, Dec. 1965, pp. 2245–2269, https://doi.org/10.1002/j.1538-7305.1965.tb04146.x.
18. Laporte, Gilbert. "The Vehicle Routing Problem: An Overview of Exact and Approximate Algorithms." *European Journal of Operational Research*, vol. 59, no. 3, June 1992, pp. 345–358, https://doi.org/10.1016/0377-2217(92)90192-c.
19. Wildenhain, M. Evan, and Ian Sacco. "Ant Colony Optimization Algorithms with Multiple Simulated Colonies Offer Potential Advantages for Solving the Traveling Salesman Problem And, by Extension, Other Optimization Problems." *Journal of Emerging Investigators*, 1 Jan. 2015, https://doi.org/10.59720/15-004.
20. Cornell University. "Dijkstra's Single-Source Shortest Path Algorithm." www.cs.cornell.edu, 2020, https://www.cs.cornell.edu/courses/cs2112/2020fa/lectures/lecture.html?id=ssp. Accessed 5 January 2024.
21. Laporte, G. "A Concise Guide to the Traveling Salesman Problem." *The Journal of the Operational Research Society*, vol. 61, no. 1, 2010, pp. 35–40.
22. Google. "Google Earth." *Google Earth*, Google, 2025, https://earth.google.com/web/.

Copyright: © 2026 Jalan and Bello-Maldonado. All JEI articles are distributed under the Creative Commons Attribution Noncommercial No Derivatives 4.0 International License. This means that you are free to share, copy, redistribute, remix, transform, or build upon the material for any purpose, provided that you credit the original author and source, include a link to the license, indicate any changes that were made, and make no representation that JEI or the original author(s) endorse you or your use of the work. The full details of the license are available at https://creativecommons.org/licenses/by-nc-nd/4.0/deed.en.