Strategic API Calls to Retrieve the Cost Matrix in the Travelling Salesman Problem

Hesam Rashidi*[1], Mehdi Nourinejad[2], and Matthew Roorda[1]

[1]Civil and Mineral Engineering, University of Toronto, Canada
[2]Civil Engineering, York University, Canada

Short summary

Last-mile logistics planners optimize delivery routes by minimizing total travel time and typically use third-party Application Programming Interfaces (APIs) for real-time travel time data. While APIs provide valuable information on traffic and road conditions, they are limited by call frequency restrictions and costs, making them expensive and time-consuming for large problem instances. This study develops a data-driven framework (requiring no pretraining) that optimizes API usage for solving the Traveling Salesman Problem (TSP). By selectively retrieving travel times for pairs of stops more likely to be part of the optimal TSP solution, the framework reduces the number of API calls, saving both runtime and API usage costs. Tested on real-world data from the 2021 Amazon-MIT Routing Challenge, the framework achieves near-optimal solutions while reducing the retrieved data to a median of 5.3% of the total matrix, with a median quality gap of 5.1% and requiring only 45.9% of the runtime.

Keywords: API usage optimization, Data-driven, Travelling Salesman Problem.

1 Introduction

Routing in last-mile deliveries is typically formulated as a Travelling Salesman Problem (TSP), where an algorithm (e.g., nearest neighbour heuristic) uses a cost matrix, such as a travel time matrix, to determine a near-optimal or optimal tour for a truck starting and ending at a depot while serving all customers. The quality of TSP solutions depends on the algorithm and the accuracy and practicality of the input cost matrix. Common approaches to generating the cost matrix for last-mile routing include (1) Euclidean distances, (2) pre-trained machine learning models, and (3) third-party API services.
Euclidean distances are computationally efficient and cost-effective but fail to capture real-world travel times, asymmetries in road networks, or dynamic conditions such as traffic or closures. Pre-trained machine learning models can offer more accurate estimates but require extensive historical data for training and may struggle to adapt to real-time traffic conditions. Third-party API services, such as Google Maps and Mapbox, provide the most comprehensive and up-to-date travel time estimates, accounting for live traffic and road conditions. However, these services come with limitations, including rate limits on requests, matrix size restrictions, and variable pricing models based on the volume of data retrieved. For example, Mapbox allows a maximum of 10 input coordinates per call and 30 calls per minute, while Google Maps permits 25 input coordinates per call and up to 600 calls per minute, with a limit of 100 elements per server-side request (Google, 2025; Mapbox, 2025). These constraints create bottlenecks in solving large-scale TSPs.
This study demonstrates that making strategic API calls can reduce both costs and processing time while still achieving near-optimal solutions to the TSP. We developed a data-driven framework that operates without requiring pretraining, making it adaptable and applicable in near real-time scenarios. Rather than retrieving the travel times for all possible pairs of stops, which is often infeasible for large problem instances, the framework prioritizes the retrieval of travel times only for those pairs of stops that are most likely to be visited consecutively in the optimal TSP solution. By focusing on high-priority connections and dynamically updating the travel time matrix, the framework reduces the reliance on exhaustive API calls while maintaining solution quality comparable to the full travel time matrix approach.
The developed framework draws inspiration from the study conducted by Emami Taba (2010), which proposed a method for finding near-optimal TSP tours using a sparse cost matrix. Their approach challenged the conventional assumption that TSP inputs must be complete graphs. They introduced several methods for generating a non-complete cost matrix and developed a modified 2-opt heuristic to find a near-optimal TSP solution using this incomplete cost matrix. They compared the quality gap and runtime of their methodology against the classical approach of solving the optimal TSP solution using a complete matrix. This study is similar to the work of Emami Taba (2010) in the sense that it retrieves

a subset of the cost matrix that is more likely to be part of the optimal TSP solution. However, we employed a different retrieval strategy and did not directly solve the TSP using an incomplete cost matrix. Instead, we used an imputation function to estimate the missing elements in the cost matrix based on the retrieved elements.

2 METHODOLOGY

Consider a complete digraph $G(V, A)$ representing a TSP instance, where $V = \{v_0, v_1, ..., v_n\}$ shows the set of depot and delivery points and an arc $a_{ij} \in A$ connects node v_i to node v_j for all $v_i, v_j \in V$. The travel time between the stops is given by a cost matrix $T = [t_{ij}]$, where t_{ij} denotes the travel time over arc $a_{ij} \in A$. Note that T is asymmetric due to directional route differences (i.e., $t_{ij} \neq t_{ji}$), non-metric (i.e., the triangle inequality may not hold), and the travel time from any delivery point i to itself is zero (i.e., $t_{ii} = 0$). A feasible TSP tour $s \in S$ is a sequence that starts at v_0 (i.e., depot), visits each node $v_i \in V \setminus \{v_0\}$ exactly once, and returns to v_0, where S denotes the set of all possible feasible tours. The optimal TSP tour, denoted by $s^* \in S$, is a feasible tour with minimal total travel time. Classical routing algorithms use a complete cost matrix (e.g., the travel time matrix T) to solve s^* for a TSP instance. In the scenario under study, it is assumed that the cost matrix T is constructed by making API calls to retrieve the travel times for the arcs. Obtaining the complete travel time matrix T for a TSP instance is often prohibitively expensive and time-consuming due to the limitations and costs associated with API services. Most APIs charge per element or matrix, with costs increasing proportionally to the problem size. Additionally, matrix size limitations often necessitate multiple API calls for larger instances, potentially increasing processing time due to rate limiting. We, to this end, developed a framework to obtain a a near-optimal TSP solution without having to construct the full cost matrix T, reducing API calls, costs, and processing time. The following explains the definitions used in the developed framework:
Definition 1. *Sampling* an arc $a_{ij} \in A$ refers to making an API call to retrieve the travel time t_{ij}, which is then recorded in a travel time matrix denoted as $\hat{T}$. The matrix $\hat{T}$ represents the sampled version of the complete travel time matrix T, where only a subset of the arcs in A have their travel times retrieved from API calls and recorded. For arcs that have not been sampled, their travel times in $\hat{T}$ are imputed based on the available sampled values. The methodology used for imputing unsampled travel times is explained in the following section. If all arcs were to be sampled, then $\hat{T}$ would be identical to the complete cost matrix T. Thus, $\hat{T}$ can be seen as a progressively refined approximation of T, depending on the number of arcs sampled.
Definition 2. The *true* length of a spanning subgraph $H \subseteq G$ (e.g., a tree or feasible TSP solution) refers to the sum of sampled travel times of the arcs in H and is shown by $L(H)$. Let $\hat{L}(H)$ represent the length of H computed using the travel times from $\hat{T}$, which may include a mix of sampled and imputed values. If all arcs in H have been sampled previously, then $\hat{L}(H) = L(H)$.
The objective is to construct $\hat{T}$ with a minimal number of sampled arcs such that a classical TSP solver can use it as the cost matrix to find a feasible sequence s, where the sampled length of s, $L(s)$, closely approximates the sampled length of the optimal sequence, $L(s^*)$. The process consists of three components: (1) the arc selection strategy, which determines which arcs to sample, (2) the imputation strategy, which estimates the missing values in $\hat{T}$ using the travel times of the sampled arcs, and (3) the stopping strategy, which establishes when $L(s)$ is sufficiently close to $L(s^*)$ without directly knowing $L(s^*)$. Figure 1 illustrates the four-step framework developed in this study, combining the mentioned components. In Step-0, the framework initializes the problem instance by constructing the Euclidean distance cost matrix, $D = [d_{ij}]$, where d_{ij} represents the Euclidean distance between v_i and v_j, and calculates the bearing angles. The bearing angles represent the direction from one location to another, expressed in degrees relative to north (i.e, 0°). These initial calculations are used for imputing unsampled arcs in $\hat{T}$ and to rank the arcs based on their likelihood of being part of the optimal TSP solution. In Step-1, a lower bound and an upper bound for the TSP instance are computed using D. These bounds are then used to sample a subset of arcs, which serve as the basis for constructing the initial $\hat{T}$. Steps-2 and -3 iteratively sample additional arcs and update $\hat{T}$ until the stopping criterion is met. At this point, the framework concludes the process, feeds the final version of $\hat{T}$ to a TSP solver. We use `OR-tool`'s Guided Local Search capped to a one-second runtime for generating the final TSP solutions Perron & Furnon (2023). Each of these steps is explained in detail in the following subsections.

Arc Selection Strategy

We use the α-nearness concept introduced by Helsgaun (2000) to rank the instance's arcs, $a_{ij} \in A$, based on their likelihood of being present in the optimal TSP solution constructed from the Euclidean distance

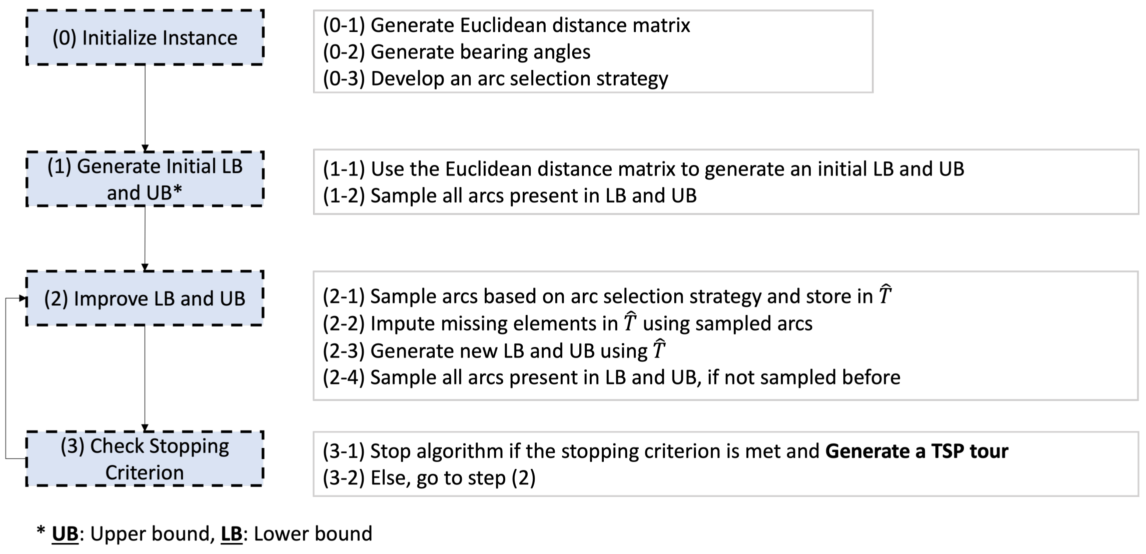

(0) Initialize Instance
(0-1) Generate Euclidean distance matrix
(0-2) Generate bearing angles
(0-3) Develop an arc selection strategy
(1) Generate Initial LB and UB*
(1-1) Use the Euclidean distance matrix to generate an initial LB and UB
(1-2) Sample all arcs present in LB and UB
(2) Improve LB and UB
(2-1) Sample arcs based on arc selection strategy and store in $\hat{T}$
(2-2) Impute missing elements in $\hat{T}$ using sampled arcs
(2-3) Generate new LB and UB using $\hat{T}$
(2-4) Sample all arcs present in LB and UB, if not sampled before
(3) Check Stopping Criterion
(3-1) Stop algorithm if the stopping criterion is met and Generate a TSP tour
(3-2) Else, go to step (2)
* UB: Upper bound, LB: Lower bound

Figure 1: The visual abstract of the developed framework.

matrix. Helsgaun (2000) constructs a spanning subgraph called a minimum one-tree, which provides a strong lower bound to the optimal TSP solution. For a symmetric complete graph such as $G(V,E)$ and an arbitrary vertex $v_i \in V$, a one-tree is defined as a spanning subgraph that connects a spanning tree on $V \setminus \{v_i\}$ to v_i using two edges incident to v_i. The minimum one-tree is the one-tree with the smallest total length. The optimal TSP sequence s^* is equivalent to the minimum one-tree where all nodes have a degree of two. Edges with a higher likelihood of being present in the minimum one-tree are more likely to be part of the optimal TSP solution (Held & Karp, 1970; Helsgaun, 2000). In our framework, we designate the depot, v_0, as the arbitrary hold-out node.
Let $F \subseteq G$ denote the minimum one-tree for G. The *nearness* of an edge $e_{ij} \in E$ connecting nodes v_i and v_j is defined as $\alpha_{ij} = L(F^+_{ij}) - L(F)$, where $L(F^+_{ij})$ is the length of the minimum one-tree required to contain e_{ij}. Larger values of α_{ij} indicate a lower likelihood of e_{ij} being part of the minimum one-tree F, and are therefore less valuable to sample. Naturally, if e_{ij} is present in F, then $\alpha_{ij} = 0$. We sort the edges based on their α_{ij} values, noting that these edges do not inherently represent the directional travel time between the two nodes they connect. Therefore, when making an API call, we sample both directions (i.e., both t_{ij} and t_{ji}). These sampled values are then stored in $\hat{T}$. In each iteration of the framework, F is calculated using $\hat{T}$ as the lower bound for the TSP, except in Steps-0 and -1, where D is used instead. We follow the methodology presented by Helsgaun (2000) to calculate the α values (i.e., the values of α_{ij} for all edges). As a benchmark, we compare the performance of the α-nearness strategy against a random arc selection strategy.

Imputation Function

An imputation function is employed to estimate the missing values using the sampled arcs. We impute the value of $\hat{t}_{ij}$, the travel time over an unsampled arc, using a linear model. This model is trained on the travel times of sampled arcs, with the Euclidean distance of the arc and its bearing angle as the independent variables. The loss function is defined as $loss = \sum_i \sum_j |t_{ij} - \hat{t}_{ij}|$. This loss function is particularly advantageous because it enables the model to be trained using linear programming, which is computationally efficient and well-suited for this type of optimization problem. The choice of independent variables is justified by two observations: (1) the travel time between two nodes is likely to increase with their Euclidean distance, and (2) traffic flow is often predominantly in one direction (towards or away from downtown or business districts). While this simple model proves effective in many scenarios, it may have limitations in cities with multiple business zones. However, delivery zones are typically batched and geographically clustered, making it unlikely for a driver to traverse multiple business districts within a single workday. For more complex urban layouts, additional dependent variables may be necessary to capture the intricacies of traffic patterns. As a benchmark, we compare the performance of the linear regression imputation strategy against a K-Nearest Neighbour (k-NN) imputation strategy (Pedregosa et al., 2011). The k-NN approach estimates missing values in the travel time matrix by identifying similar arcs (i.e., neighbours) based on their Euclidean distance and bearing angle. For each missing value, the k-NN strategy uses the n-nearest known values (e.g., five), weighting their contributions based on proximity, to generate an estimate.

Stopping Criterion

The framework employs a hybrid stopping criterion to determine when the iterative process of improving $\hat{T}$ should end. This approach combines two complementary strategies. The first strategy monitors changes in the *sampled* length of upper bound and lower bound between iterations. Using the Nearest Neighbour (NN) heuristic as an upper bound for the TSP, the framework tracks the relative changes in these bounds. If both the upper bound and lower bound stabilize, with their relative changes falling below a predefined threshold (e.g., 5%), it indicates that $\hat{T}$ contains the most likely arcs to be within the optimal TSP tour. However, if either bound continues to change more than the set threshold, it suggests that $\hat{T}$ is still improving, and the process should continue. Once the threshold is reached, the framework performs 10 additional iterations to ensure stability before stopping. Both upper bound and lower bound are computed using $\hat{T}$ as the cost matrix, except during initialization (Steps 0 and 1), where the Euclidean distance matrix (D) is used. The sampled lengths of upper bound and lower bound are considered to check the stopping criterion. The threshold and additional iterations can be adjusted based on user requirements. The second strategy imposes a limit on the total number of API calls, which ensures that no more than half of the travel time matrix is retrieved. If this limit is reached, the framework transitions from strategic sampling to batch retrieval of travel times. This switch acknowledges that irregular or unexpected travel time patterns can reduce the effectiveness of strategic sampling.
Once the framework determines that the stopping criteria are met, the constructed $\hat{T}$ is passed to `OR-Tools` to solve the TSP instance. The sampled length of the resulting sequence is then used to evaluate the quality of the solution. This study was implemented using a Macbook Air with an eight-core Apple M2 chip and 16 GB of RAM.

3 RESULTS AND DISCUSSION

This study uses the open-source dataset from Amazon's Last-Mile Research Challenge, which contains 6,112 historically realized TSP instances in five cities in the United States (Merchán et al., 2022). Each delivery instance in the dataset includes the coordinates of each stop and the travel times between pairs of stops. The travel times are based on real-world conditions, meaning they are not symmetric and in some cases do not satisfy the triangle inequality. These travel times represent the duration between stops and do not account for service times at the stops. For descriptive statistics, refer to Merchán et al. (2022).
Figure 2 illustrates a sample problem instance to demonstrate how the framework operates. The line plots with markers illustrate the sampled values of the lower bound (blue), upper bound (red), and the TSP tour duration (black) computed by using $\hat{T}$ (i.e., $\hat{T}$ was used to generate the mentioned subgraphs and then their arcs were sampled to retrieve their true length). The dashed lines represent the true values of the lower bound, upper bound, and optimal TSP tour duration, calculated using the fully sampled travel time matrix T. The dashed lines are not calculated as part of the framework and are included only for demonstration. According to Figure 2, after 12 iterations, the quality gap between the solution generated by the framework and the optimal TSP solution computed using T is 5.3%. Notably, only 1,328 arcs are sampled for a problem size of 166 stops (i.e., 4.8% of arcs were sampled). With each iteration, $\hat{T}$ contains more sampled elements, leading to convergence of the upper and lower bound values to their true values, as shown in Figure 2. The figure shows that the sampled length of the upper and lower bounds fluctuate. These small fluctuations are due to the data-driven imputation methodology, which inherently introduces some error.
The results of the algorithms are summarized in Figure 3, which compares their performance across three metrics: (a) runtime relative to the baseline, (b) quality gap compared to the baseline, and (c) the number of arcs sampled as a ratio of the total possible arcs. These results are based on a simulation of retrieving arcs from the Google Maps Travel Time API, taking into account the service limitations, such as rate limits and processing time, to reflect realistic constraints associated with API usage (Google, 2025). The baseline represents the case where the optimal TSP sequence is generated using the complete travel time matrix. The "Euclidean" algorithm uses the Euclidean distance matrix to generate the optimal TSP sequence, followed by sampling its arcs. The other algorithms explore variations of the proposed methodology, combining different arc sampling strategies (α-nearness and random selection) and imputation methods (linear regression and k-NN). Notably, the method with α-nearness guiding its sampling strategy with the linear regression imputation function, which is bolded in the plots, represents the proposed combination.

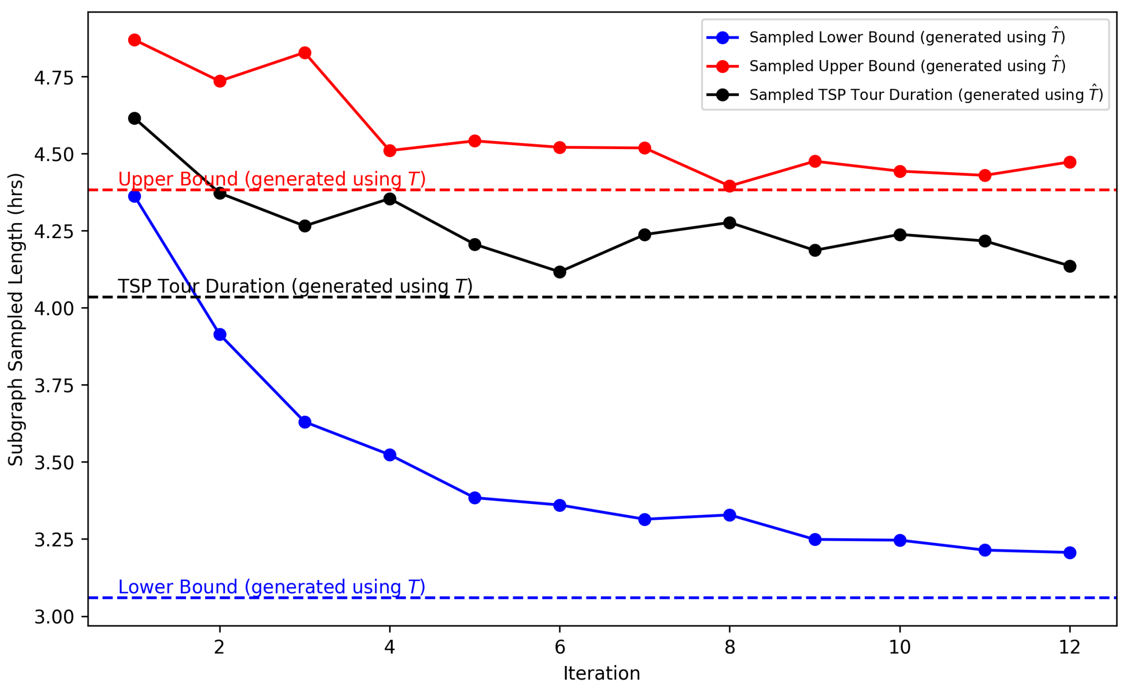
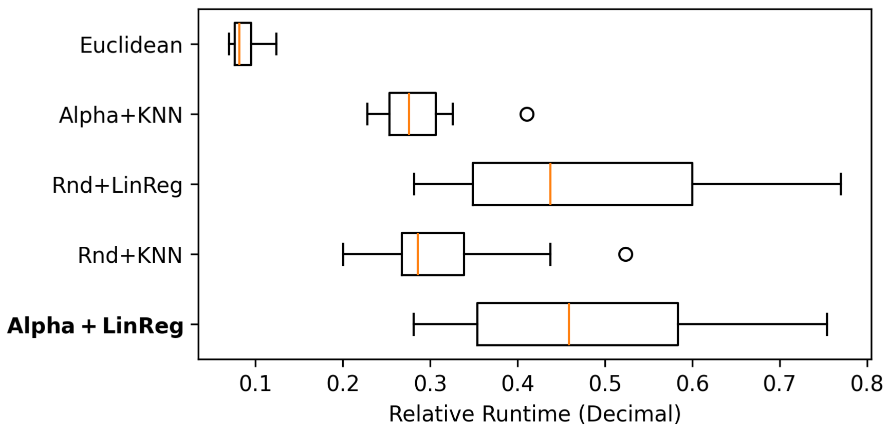


Figure 2: Framework applied to a sample problem instance. The line plots with markers illustrate the sampled values of the lower bound (blue), upper bound (red), and optimal TSP solution (black) using the imputed travel time matrix ($\hat{T}$) in each iteration (i.e., L(sub graph)). The dashed lines represent the actual values of the lower bound (blue), upper bound (red), and optimal TSP tour (black), computed using the fully sampled travel time matrix (T).

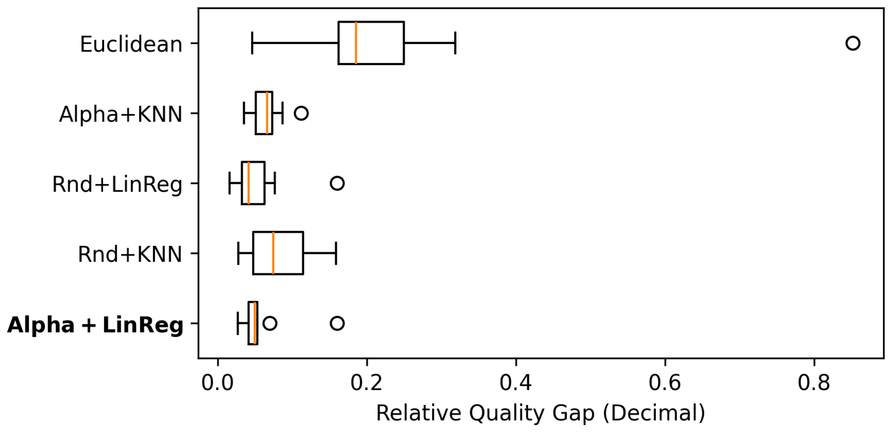
Figure 3a shows the runtime comparison of the algorithms relative to the baseline. The Euclidean algorithm is the fastest and requires sampling the least number of arcs (see Figure 3c), with a median runtime of 8.1% of the baseline. As expected, however, its performance in terms of quality is suboptimal. As shown in Figure 3b, the Euclidean approach has the largest quality gap among all algorithms, with a median value of 18.6%.
Figure 3b demonstrates that all combinations of arc sampling strategies and imputation methods improve the quality gap compared to the Euclidean method. In the worst-case scenario, the *Rnd+KNN* method (random sampling with k-NN imputation) has a median quality gap of 7.5%, while the best results are achieved by the *Alpha+LinReg* method (α-nearness sampling strategy with linear regression imputation), with a median quality gap of 5.1%. These two cases respectively sampled a median of \$4.7% and 5.3% of the entire travel time matrix. In terms of runtime, Figure3a highlights that *Rnd+KNN* and *Alpha+LinReg* require a median of 28.6% and 45.9% of the runtime compared to the case where the complete travel time matrix is retrieved.

4 Conclusions

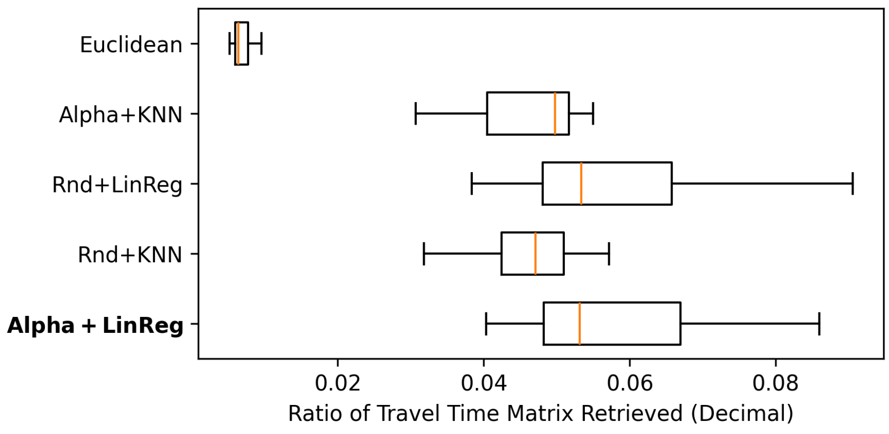
Third-party services, such as Google Maps, provide real-time data on travel conditions— including live traffic, road closures, and other dynamic factors that influence travel times. Users can access this information through Application Programming Interface (API) calls. Retrieving the travel time matrix through APIs for solving routing problems, such as the Travelling Salesman Problem (TSP), improves the practicality and reliability of the solutions. However, obtaining the complete travel time matrix through API calls is both costly and time-consuming. APIs typically charge based on the size of the matrix or the number of elements retrieved, with costs increasing as the problem size increases. Additionally, matrix size limitations often necessitate breaking larger requests into multiple API calls. This process is further constrained by rate limits on the number of calls that can be made per minute, which increases processing time for larger problem instances. This study demonstrates that making strategic API calls can reduce costs and processing time while achieving near-optimal TSP solutions. We developed a data-driven framework that operates without the need for pretraining and can be applied in near real-time. Instead of retrieving the travel times for all pairs of stops, the framework selectively retrieves travel times



(a) Comparison of algorithm runtimes relative to the baseline, where the optimal sequence is generated using the complete travel time matrix.



(b) Comparison of algorithm quality gaps, with the baseline being the case where the optimal sequence is generated using the complete travel time matrix.



(c) Comparison of the number of arcs sampled as a ratio of the total possible arcs.

Figure 3: The comparison of the algorithms based on (a) runtime, (b) quality gap, and (c) the number of arcs sampled. The baseline refers to the case where the optimal TSP sequence is generated using the complete travel time matrix. The *Euclidean* algorithm generates the optimal TSP sequence using the Euclidean distance matrix, after which its arcs are sampled. The other algorithms represent variations of the proposed methodology, employing different arc sampling strategies (denoted as *Alpha* for method of α-nearness and *Rnd* for random selection) and imputation methods (*LinReg* for linear regression and *KNN* for k-Nearest Neighbours). The *Alpha+LinReg* combination (bolded in the plots) represents the proposed combination methodology.

only for the pairs of stops that are more likely to be visited consecutively in the optimal TSP solution. The baseline for comparison is the case where the full travel time matrix is retrieved, providing the optimal TSP solution. The proposed framework reduces the number of retrieved travel times to a median of 5.3% of the total matrix. This comes with a median quality gap of 5.1% and requires a median of 45.9% of the runtime compared to the baseline.

Acknowledgements

Funding for this research was provided by the Natural Science and Engineering Research Council of Canada. The grammar of this study was reviewed using GPT-4, which provided suggestions that were manually evaluated and incorporated into the revisions of the paper.

References

Emami Taba, M. S. (2010). *Solving traveling salesman problem with a non-complete graph* (Unpublished master's thesis). University of Waterloo.

Google. (2025). Author. Retrieved from `https://developers.google.com/maps/documentation/distance-matrix/usage-and-billing?_gl=1%2A1lhccji%2A_up%2AMQ..%2A_ga%2ANTUyNjg0OTcyLjE3MzcyMzA1ODk.%2A_ga_NRWSTWS78N%2AMTczNzIzMDU4OS4xLjEuMTczNzIzMDU5NS4wLjAuMA..`

Held, M., & Karp, R. M. (1970). The traveling-salesman problem and minimum spanning trees. *Operations Research*, *18*. doi: 10.1287/opre.18.6.1138

Helsgaun, K. (2000, 10). An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, *126*, 106-130. doi: 10.1016/S0377-2217(99)00284-2

Mapbox. (2025). Author. Retrieved from `https://docs.mapbox.com/api/navigation/matrix/`

Merchán, D., Arora, J., Pachon, J., Konduri, K., Winkenbach, M., Parks, S., ... Merchán, M. (2022). Amazon last mile routing research challenge: Data set. *Transportation Science*. doi: 10.1287/trsc.2022.1173

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Perron, L., & Furnon, V. (2023). *Or-tools*. Retrieved from `https://developers.google.com/optimization`