Scientific Research Publishing

American Journal of Operations Research, 2024, 14, 169-190
https://www.scirp.org/journal/ajor
ISSN Online: 2160-8849
ISSN Print: 2160-8830

An Elementary Approach to the Vehicle Routing Problem via Python and Google API

Kai Q. Lian, Gareth A. Tribello

Centre for Quantum Materials and Technologies, School of Mathematics and Physics, Queen's University Belfast, Northern Ireland, UK
Email: klian03@qub.ac.uk, g.tribello@qub.ac.uk

How to cite this paper: Lian, K.Q. and Tribello, G.A. (2024) An Elementary Approach to the Vehicle Routing Problem via Python and Google API. *American Journal of Operations Research*, **14**, 169-190.
https://doi.org/10.4236/ajor.2024.146009

Received: November 2, 2024
Accepted: November 26, 2024
Published: November 29, 2024

Copyright © 2024 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

Commercial organisations commonly use operational research tools to solve vehicle routing problems. This practice is less commonplace in charity and voluntary organisations. In this paper, we provide an elementary approach for solving the Vehicle Routing Problem (VRP) that we believe can be easily implemented in these types of organisations. The proposed model leverages mixed integer linear programming to optimize the pickup sequence of all customers, each with distinct time windows and locations, transporting them to a final destination using a fleet of vehicles. To ensure ease of implementation, the model utilises Python, a user-friendly programming language, and integrates with the Google Maps API, which simplifies data input by eliminating the need for manual entry of travel times between locations. Troubleshooting methods are incorporated into the model design to ensure easy debugging of the model's infeasibilities. Additionally, a computation time analysis is conducted to evaluate the efficiency of the code. A node partitioning approach is also discussed, which aims to reduce computational times, especially when handling larger datasets, ensuring this model is realistic and practical for real-world application. By implementing this optimized routing strategy, logistics companies or organisations can expect significant improvements in their day-to-day operations, with minimal computational cost or need for specialised expertise. This includes reduced travel times, minimized fuel consumption, and thus lower operational costs, while ensuring punctuality and meeting the demands of all passengers.

Keywords

Vehicle Routing Problem (VRP), Google Maps Distance Matrix API, Python, Pulp, Mixed Integer Linear Programming, Transportation, Optimisation Problem, Time Window, Mathematical Modelling

1. Introduction

This paper originates from a project undertaken for the final-year module, Stochastic Processes and Risk in the BSc Mathematics undergraduate program at Queen's University Belfast. The project was developed in collaboration with Fermanagh Community Transport (FCT) [1], which is a transport service provider to individuals in rural and socially isolated areas in and around Fermanagh. Fermanagh is a largely rural county located in Northern Ireland, which is composed of dispersed communities, thus making transportation services, like those provided by FCT, essential for connecting residents to essential services and resources. Their services, such as the Dial-A-Lift scheme and the Home to Hospital transport service, cater primarily to the elderly, people with disabilities, and individuals facing mobility challenges. These services are vital in enabling people who lack access to public or private transport options to maintain their independence and attend essential appointments, especially health and well-being visits. FCT uses a fleet of minibuses and volunteer-driven cars to facilitate these operations. However, due to the rural and widespread nature of the service area, assigning efficient vehicle routes is a crucial aspect of maintaining a cost-effective and reliable transport service. Hence, this project proposes a solution for determining the optimal sequence for picking up customers from various locations and transporting them to a final destination, tailored specifically to FCT's operational needs through mathematical modelling. In this paper, the original FCT-focused project has been significantly revised and expanded to serve as a practical model for general applications in vehicle routing and logistics. This broader approach seeks to provide a versatile framework that can be adapted to various real-world scenarios, delivering both practical insights and impactful solutions in transportation systems for volunteer or charity logistics organisations with limited computational resources.

The vehicle routing problem (VRP) is considered among the most challenging in combinatorial optimization [2] and is NP-hard, because it encompasses the Travelling Salesman Problem (TSP) as a specific case [3]. In recent years, the evolution of VRP has focused on adapting VRP to real-world applications, resulting in new VRP variants, models, solution techniques, and practical implementations. The VRP is a fundamental problem in logistics and operational research. It involves designing optimal delivery or collection routes from a central depot to a set of geographically dispersed customers while adhering to constraints such as vehicle capacity, route length, time windows, and precedence relations [3]. The objective in solving VRP is to minimize the total cost associated with these routes, generally due to factors such as distance travelled, fuel consumption, vehicle utilization, and driver costs. The Vehicle Routing Problem with Time Windows (VRPTW) is one of the popular extensions of this problem, which assumes that each customer must be visited within a specific time interval [4]. The General Vehicle Routing Problem (GVRP) is a complex extension of the classic VRP that incorporates various real-life constraints and requirements, such as time windows, heterogeneous fleets, multi-dimensional capacity constraints, and order/vehicle compatibility [5].

These extensions make GVRP a more realistic and challenging problem to solve, but also a more powerful tool for optimizing complex real-world logistics operations.

The VRP is encountered daily by numerous distributors worldwide and carries substantial economic importance as it directly impacts operational costs and service efficiency. Common applications arise in newspaper industries, food delivery and milk collection services [3]. The primary advantage of employing VRP solutions is operational cost reduction. By optimizing routes, businesses or organisations can minimize the total distance travelled by their vehicles, leading to lower fuel consumption and reduced vehicle wear and tear. Also, VRP solutions boost customer satisfaction by enabling timely and reliable deliveries. Their adaptability to real-world constraints such as vehicle capacity, route length, time windows, and specific customer requirements allows businesses to tailor VRP models to their unique operational needs, maximizing efficiency and cost-effectiveness. Additionally, in the context of environmental sustainability, VRP minimizes travel distances and fuel consumption, directly reducing greenhouse gas emissions and supporting the development of greener logistics systems [2]. The emerging focus on green VRP expands traditional VRP models to explicitly consider environmental impacts, incorporating constraints related to energy efficiency and fuel usage. This shift is increasingly important for organisations aiming to meet sustainability goals without compromising service quality.

Unlike more complex vehicle routing problem (VRP) solutions, which will be discussed in Section 4.4, that typically demand considerable computational power and specialized knowledge, this paper discusses an approach which strikes a practical balance between simplicity, cost-effectiveness, and adaptability, making it particularly suitable for smaller-scale volunteer or charity operations like FCT. Advanced VRP methodologies often require complex algorithms and extensive data analysis capabilities, which can be overwhelming and impractical for organisations lacking the resources to implement such solutions. In contrast, this model's straightforward approach allows smaller organisations to effectively tackle routing challenges without the need for extensive technical expertise or substantial investments in computational resources. The implementation of the model proposed in this paper is facilitated through user-friendly tools such as the Google Maps Distance Matrix API [6] and Python programming language, which enhance its accessibility and usability. The Google Maps API provides real-time data on traffic conditions, distances, and estimated travel times, allowing users to generate optimal routes efficiently. Python, known for its simplicity and versatility, enables quick development and customization of the routing model. This combination results in a simple user interface that allows staff members or users with minimal technical training to engage with the routing solution effectively, reducing the learning curve typically associated with more advanced VRP systems. Ultimately, the methodology and implementation explored in this paper aim to provide a valuable framework for improving transportation logistics in a variety of real-world

contexts, in a straight forward, yet impactful and effective manner.

2. Methodology

In this section, we describe the mathematical modelling of the Vehicle Routing Problem (VRP). In our model, various locations will be regarded as nodes, each representing a specific point on the route. The first node will mark the departure point, often referred to as the central hub or depot, where the fleet of vehicles initiates its journey. The last node represents the final destination or drop-off point, that is, the terminating node for each vehicle's route. This structure will allow for a clear and consistent reference framework as we discuss routing and optimization within the VRP context.

2.1. Assumptions

The model is formulated based on the following assumptions:

- All vehicles depart from the same node, central hub.
- All vehicles end the trip at the same node, a final destination.
- The customers' demand at every node must be satisfied.
- Each customer node must be visited exactly once by a vehicle.
- The maximum number of vehicles available for service is n_k. Note that the optimal solutions may not require all these vehicles to be used.
- Each vehicle has the same fixed capacity of Q, that is the total demand of the customers visited by a single vehicle in a trip must not exceed Q.
- The arrival time of the vehicles must satisfy all customers' specified time window. All vehicles are allowed to reach nodes earlier than the specified time window, but the waiting time is accounted for in the total route time.
- The total routing time of each vehicle in a single trip must not exceed T, to ensure all customers are able to reach their destination on time.

2.2. Parameters and Sets

The notations used in the model are as follows:

- n: The total number of nodes in the system.
- n_k: The maximum number of vehicles available for routing.
- $nodes = \{i : i = 0,1,2,\cdots,n-1\}$: The set of all nodes in the system, where $i = 0$ represents the central hub, $i = 1$ to $i = n-2$ are the customer nodes, and $i = n-1$ is the final drop station.
- $vehicles = \{k : k = 0,1,2,\cdots,n_k - 1\}$: The set of all available vehicles.
- Q: The capacity of each vehicle.
- M: An arbitrarily large positive integer, used for sequencing logic.
- d_i: The demand at customer node i, where $i \in \{1,2,\cdots,n-2\}$. The central hub and drop node have zero demand (*i.e.*, $d_0 = 0$ and $d_{n-1} = 0$).
- e_i: The earliest permissible arrival time at node i, where $i \in \{1,2,\cdots,n-2\}$.
- l_i: The latest permissible arrival time at node i, where $i \in \{1,2,\cdots,n-2\}$.
- t_{ij}: The travel time between node i and node j, where $i, j \in \{0,1,\cdots,n-1\}$.

- T: The maximum total routing time of each vehicle in a single trip.

2.3. Decision Variables

The variables implemented in the system are as follows,

- $x_{ijk} \in \{0,1\}$: A binary decision variable that equals 1 if vehicle k travels from node i to node j, and 0 otherwise.
- $z_{ik} \in \{0,1\}$: A binary decision variable that equals 1 if vehicle k visits node i, and 0 otherwise.
- $u_k \in \{0,1\}$: A binary decision variable that equals 1 if vehicle k is used in the routing plan, and 0 otherwise.
- $a_{ik} \in \mathbb{R}^+$: A continuous variable that represents the arrival time of vehicle k at node i.
- $\max_a_k \in \mathbb{R}^+$: A continuous variable that represents the maximum arrival time for each vehicle k, *i.e.*, the arrival time at the final drop station for vehicle k, *i.e.*, the total routing time of each vehicle k.

2.4. Objective Function

The objective of the model is to minimize the total travel time for all vehicles, as we assume the total costs incurred are directly proportional to the total travel time of all vehicles. The objective function can be expressed as:

$$\min z = \sum_{k=0}^{n_k-1} \max_a_k \tag{O1}$$

2.5. Constraints

- **Initial and Final Conditions:** Every vehicle departs from the central hub at time 0 and visits the final drop node. The variable u_k in the subsequent equations ensures that the constraints apply only to the vehicles that are utilised.

$$a_{0k} = 0,\ z_{0k} = u_k,\ z_{n-1,k} = u_k \quad \forall k \in vehicles \tag{C1}$$

- **End Node Constraint:** The arrival time at the final drop node must be greater than or equal to the arrival times at all customer nodes for each vehicle, ensuring that all vehicles conclude their routes at the drop station. The arbitrarily large constant, M enables the termination of the following inequality if vehicle k does not visit node i, by making the corresponding side of the inequality below to be arbitrarily small.

$$a_{n-1,k} \ge a_{ik} - M\left(1 - z_{ik}\right) \quad \forall i \in \{1,2,\cdots,n-2\}, \forall k \in vehicles \tag{C2}$$

- **Visit Constraint:** Each customer node is visited exactly once by one vehicle.

$$\sum_{k=0}^{n_k-1} z_{ik} = 1 \quad \forall i \in \{1,2,\cdots,n-2\} \tag{C3}$$

- **Capacity Constraint:** The total demand served by any vehicle in a trip must not exceed its capacity. Similar to Equation (C1), the variable u_k ensures that the constraint applies only to the vehicles that are utilised.

$$\sum_{i \in nodes} d_i z_{ik} \le Q u_k \quad \forall k \in vehicles \tag{C4}$$

- **Time Flow and Sequence Logic:** The following inequality ensures that if vehicle k travels from node i to node j, the arrival time at node j is at least the arrival time at node i plus the travel time between the two nodes. Similar to Equation (C2), the arbitrary large constant M makes the inequality ineffective if vehicle k does not travel from node i to node j.

$$a_{jk} + M\left(1 - x_{ijk}\right) \ge a_{ik} + t_{ij} - M\left(2 - z_{ik} - z_{jk}\right) \quad \forall i, j \in nodes, \forall k \in vehicles \tag{C5}$$

Also, the following constraint ensures that vehicles can only travel between nodes if they visit both nodes:

$$x_{ijk} + x_{jik} \ge 1 + M\left(z_{ik} + z_{jk} - 2\right) \quad \forall i, j \in nodes, \forall k \in vehicles \tag{C6}$$

- **Time Window Constraint:** The arrival time at each customer node must lie within its specified time window. Again, the variable z_{ik} ensures that the constraint applies only to vehicle k which visits node i.

$$e_i z_{ik} \le a_{ik} \le l_i z_{ik} \quad \forall i \in nodes, \forall k \in vehicles \tag{C7}$$

- **Total Routing Time Constraint:** The maximum arrival time (*i.e.*, the total routing time) for each vehicle k must not exceed T.

$$T \ge a_{n-1,k} \quad \forall k \in vehicles \tag{C8}$$

- **Objective Variable:** The maximum arrival time (*i.e.*, the total routing time) for each vehicle k is equal to the arrival time at the final drop node.

$$\max_a_k = a_{n-1,k} \quad \forall k \in vehicles \tag{C9}$$

3. Implementation and Results

In this section, we discuss the use of Google Maps API to determine the travel time between locations. Also, we demonstrate the implementation of the mathematical model using a predetermined set of parameters, specifically chosen to mimic the day-to-day operation of Fermanagh Transport Community [1]. Note that this approach is not only tailored to FCT's specific needs but can also be changed by other logistics organisations, allowing for flexibility and applicability across different operational contexts.

3.1. Travel Time via Google Maps API

For the travel time parameter, we utilized the Google Maps Distance Matrix API [6] to efficiently determine the travel time matrix, t_{ij} between all nodes instead of manually calculating these times. This API allows us to predict the travel times for a particular day and time on which we intend to travel. By considering future traffic patterns that vary throughout the day, we can obtain more accurate results that reflect the expected travel times. Additionally, the service time at each node can be simply added to each element of the matrix t_{ij}, depending on the circumstances, thus providing a more realistic representation of the total travel time. The nodes, represent specific locations, that is, each customer's departure and drop-

off addresses.

The complete Python script used for this demonstration is given in Appendix A.1.2. The travel times for the API call are set to *8 a.m.* on *25th October 2024*. Nine specific locations in County Fermanagh, Northern Ireland have been selected as nodes, with Ballinamallard designated as the departure point and Enniskillen as the drop-off station. The selected nodes are as follows:

- **Node 0:** Ballinamallard, Enniskillen BT94 2FA
- **Node 1:** Belleek, Enniskillen BT93 3FY
- **Node 2:** Brookeborough, Enniskillen BT94 4EY
- **Node 3:** Clabby, Fivemiletown BT75 0QZ
- **Node 4:** Derrygonnelly, Enniskillen BT93 6GA
- **Node 5:** Irvinestown, Enniskillen BT94 1EN
- **Node 6:** Maguiresbridge, Enniskillen BT94 4RY
- **Node 7:** Lisnaskea, Enniskillen BT92 0FL
- **Node 8:** Fairgreen Shopping Centre, Forthill St, Enniskillen BT74 6JA

The travel time matrix, t_{ij} associated with these 9 nodes is evaluated and shown in **Table 1**. Note that the diagonal elements of the travel time matrix t_{ij} are zero, as the time taken to travel from node i to itself is inherently zero. By implementing the API, we obtain accurate and up-to-date travel times based on real-world traffic conditions, while also reducing the potential for human errors associated with manual time calculation. The travel times obtained from the API are subsequently integrated as a parameter into the vehicle routing model, facilitating a more efficient optimization process.

Table 1. Travel time matrix (in minutes) showing the estimated travel times between the nine nodes, as calculated using the Google Maps Distance Matrix API. Each entry x_{ij} represents the travel time from node i to node j.

	Node 0	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Node 0	0.00	41.22	23.83	20.47	27.30	9.48	21.73	28.28	11.38
Node 1	40.48	0.00	48.73	52.52	23.33	33.13	45.22	51.75	34.78
Node 2	24.42	48.10	0.00	10.67	31.93	28.07	4.52	8.55	15.50
Node 3	20.27	51.92	10.62	0.00	35.75	23.75	14.07	19.17	18.78
Node 4	25.38	22.98	31.60	35.38	0.00	29.58	28.08	34.63	17.67
Node 5	8.85	32.98	27.55	23.07	30.83	0.00	25.45	32.00	14.90
Node 6	22.47	44.58	4.62	14.05	28.42	26.67	0.00	6.55	11.98
Node 7	28.88	51.02	8.57	18.98	34.83	33.10	6.43	0.00	18.42
Node 8	10.48	33.95	15.23	18.57	17.78	14.68	11.72	18.27	0.00

3.2. Setting Parameters

We first define all parameters used in the model before demonstrating the results. We use the travel time matrix, t_{ij} as defined in **Table 1**, *i.e.*, we have a total of $n = 9$ nodes. Then, we set the number of available vehicles $n_k = 3$, each with a

maximum capacity of $Q = 12$. The maximum arrival time for each vehicle is set to $T = 90$ minutes. The demands, d_i and the starts, e_i, and ends, l_l of the time windows for each node are as follows,

$$d_i = [0,3,2,1,2,3,2,1,0]$$

$$e_i = [0,50,40,30,20,35,45,55,0]$$

$$l_i = [M,60,50,40,30,45,55,65,M]$$

Note that the array of demands, d_i represents the demand at each node $i \in \{0, n-1\}$, hence d_0 and d_{n-1} are zero since there are no customers at the starting and ending points. Also, e_i and l_i representing the earliest and latest arrival time at each node $i \in \{0, n-1\}$, is set to be 10 minutes apart for the customer nodes. Meanwhile, the latest arrival time at the start and final nodes are set to $M = 10000$, an arbitrarily large integer to allow the time windows at these nodes to be arbitrarily wide.

3.3. Solution Using PuLP

The mathematical model is solved using the PuLP library in Python, which provides an interface to linear programming solvers. The complete Python script used for the linear optimization is listed in **Appendix A.1.1**. For the output results, instead of printing out a list of all optimised decision variables and manually inspecting each variable, the following approach processes the results and displays the actual sequence of node visits for each vehicle in an easily readable and understandable format.

First, the status of the optimization model, *i.e.*, whether it is infeasible or optimal, and the total cost of the solution, *i.e.*, the value of the objective function *z*, are returned. It then collects the nodes visited by each vehicle, along with their respective arrival times. The nodes are then sorted by arrival time for each vehicle to match the actual route taken by each vehicle. This information is stored in a dictionary, where each vehicle is associated with its corresponding route. Also, the waiting time at each customer node is calculated by subtracting the sum of the previous arrival time and the travel time to that node from the arrival time at that customer node. Finally, the code generates a detailed route description for each vehicle, including the arrival time and the waiting time at each node, presented in sequence. From **Figure 1**, we see that the optimization returned an optimal status with a total routing time of 208.38 minutes across all vehicles, and a detailed route assignment for each vehicle.

```
Total Cost =  208.38
Vehicle 0 moves from node 0 (arrives at time 0.00, waits for 0.00 min) -> node 3 (arrives at time 20.47, waits for 9.53 min)
-> node 2 (arrives at time 40.62, waits for 0.00 min) -> node 6 (arrives at time 45.14, waits for 0.00 min)
-> node 7 (arrives at time 51.69, waits for 3.31 min) -> node 8 (arrives at time 73.42, waits for 0.00 min)
Vehicle 1 moves from node 0 (arrives at time 0.00, waits for 0.00 min) -> node 5 (arrives at time 9.48, waits for 25.52 min)
-> node 8 (arrives at time 49.90, waits for 0.00 min)
Vehicle 2 moves from node 0 (arrives at time 0.00, waits for 0.00 min) -> node 4 (arrives at time 27.30, waits for 0.00 min)
-> node 1 (arrives at time 50.28, waits for 0.00 min) -> node 8 (arrives at time 85.06, waits for 0.00 min)
```

Figure 1. Vehicle routing output.

This method of printing the result gives a more insightful view of vehicle routes than simply reviewing the raw decision variables which is tedious and time consuming. While the output of the script is returned in only 4.239 seconds, the computation time may increase significantly with larger node counts; this matter will be discussed in Section 4.2 & 4.3.

3.4. Visualization of Solution

To plot visual representations of the solution, we utilized the matplotlib, NetworkX and Folium libraries in Python, resulting in two figures that illustrate the vehicle routing solution. Both plots display the nodes, including the central hub, customer locations, and the final drop-off station, with each vehicle's route highlighted in distinct colours. The lines connecting the nodes represent the optimal sequence for picking up the customers associated with each vehicle. The arrival and waiting times, demands, and time windows at each node are annotated, providing a concise and descriptive routing solution that makes it easy for the user to determine whether a solution has been found which adheres to the inputted constraints.

Figure 2 presents a detailed visual plot of the optimal route that was found by the algorithm. Each node in this figure represents a location with specified customer demands and time windows, showing that all vehicles depart from Ballinamallard and end at the drop-off station in Enniskillen, with arrival times clearly satisfying the maximum routing time of $T = 90$ minutes. Additionally, Figure 3 offers a geographical context for the routing solution, showcasing the vehicle routes and nodes within the actual locations of customers and drop-off points, thereby further clarifying the distances involved in the routing problem.

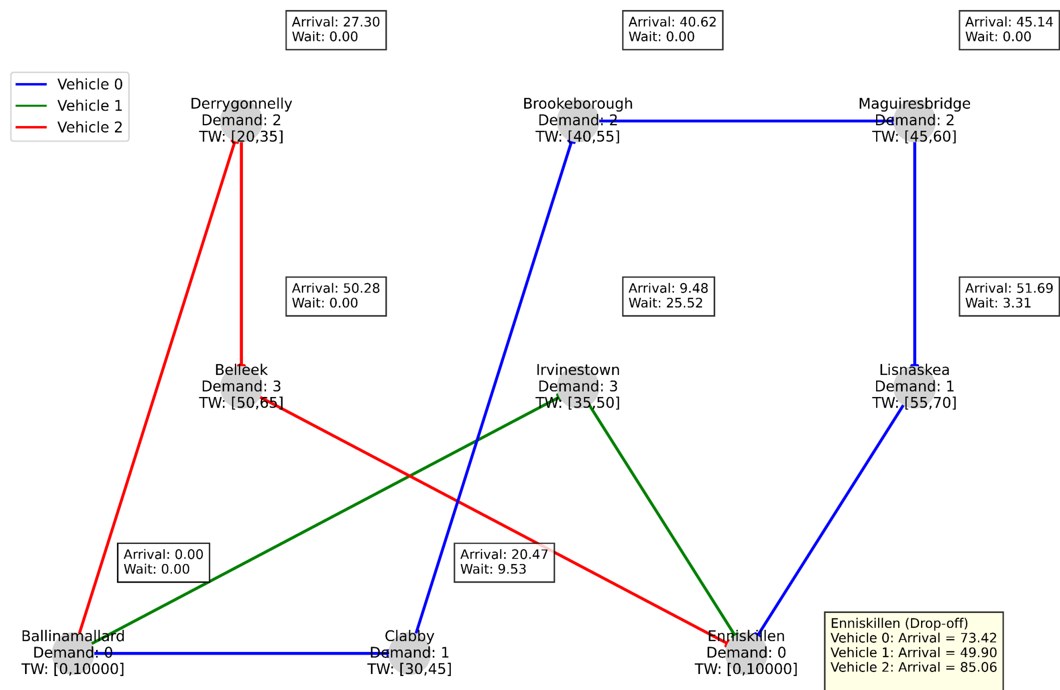

Figure 2. Vehicle routing solution on a graph (Note that the graph is not drawn to scale).

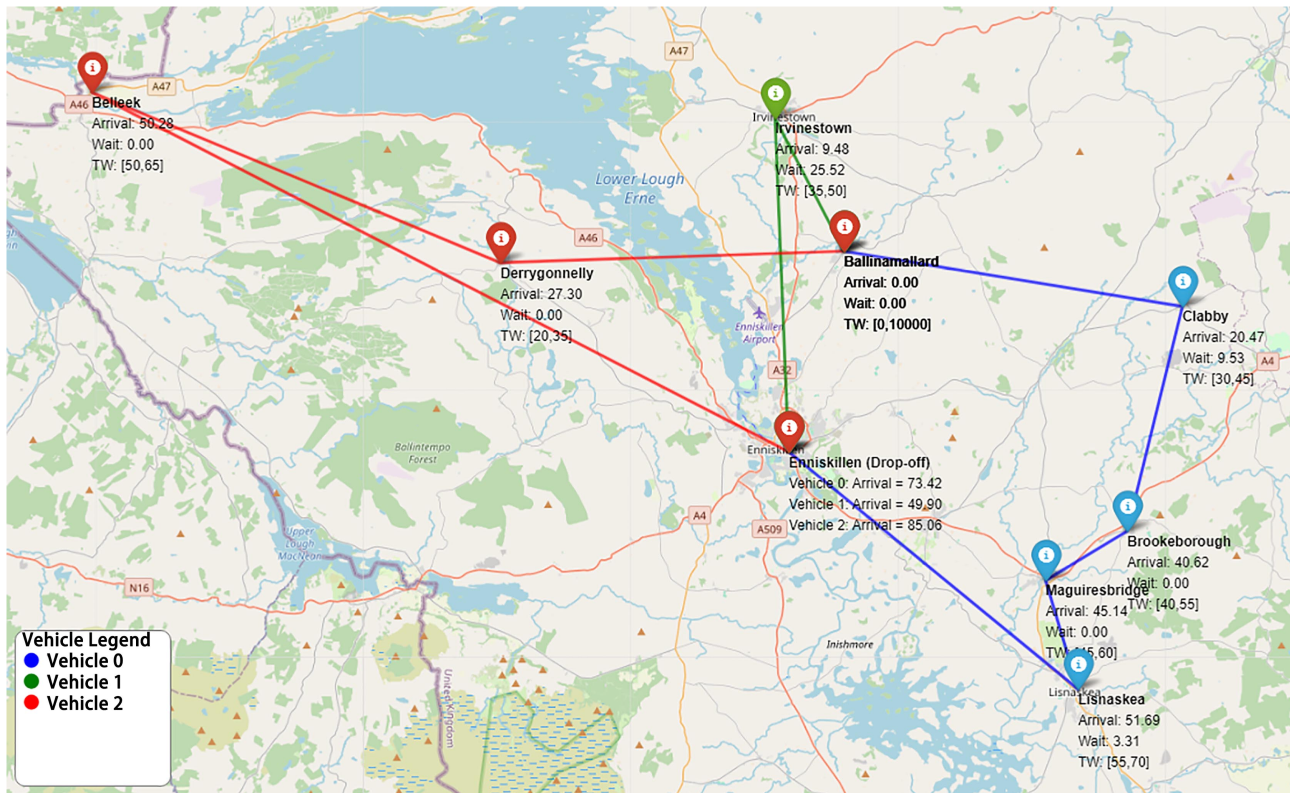

Figure 3. Vehicle routing solution on a map.

3.5. Troubleshooting Infeasible Status

In order to efficiently identify the causes of infeasibility in our vehicle routing model, a straightforward troubleshooting system is implemented. When the optimization model returns an infeasible status, the constraints that cannot be satisfied simultaneously are indicated. There are two primary potential causes of this infeasibility, namely time window and vehicle capacity violations.

Time window violations occur due to several factors. One common issue arises from the total time horizon, T, being insufficient to transport all of the customers. If the time allowed for service, T is too short, and the vehicles may be unable to reach all nodes within their designated time frames. Additionally, if the time windows for customers are too concentrated and tight, it can create scheduling conflicts where the vehicles cannot feasibly travel to the nodes in time to meet their respective windows. On the other hand, vehicle capacity violations can occur if the total demand from the customers assigned to a vehicle exceeds its maximum capacity.

Our troubleshooting implementation aims to quickly assess these two key issues. If a vehicle capacity violation is detected, the system outputs a clear message stating, "Vehicle capacity not satisfied." If capacity is not the issue, the system then checks for time window violations and returns the message, "Time window not satisfied." By simplifying the identification of key issues, this aids the user in adjusting the parameters accordingly, allowing the system to return a feasible solution.

4. Discussion

In this section, we discuss the model weaknesses, as well as the measures that must be taken to ensure that this mathematical model can be used in real-world scenarios, where the datasets can be arbitrarily large. A computation time analysis is also conducted to evaluate the efficiency and applicability of this model with different numbers of nodes. Finally, we discuss existing VRP models developed by other researchers and organisations.

4.1. Model's Limitations

There are a few limitations to this model. Firstly, this model is designed to accommodate only a single destination, which may not be sufficient for organisations with more complex routing needs. Additionally, the model does not incorporate AND/OR precedence constraints [4], which can be critical for situations where certain locations must be visited together or in a specific order. The implementation of the precedence constraints will be addressed in Section 4.4.2. Also, we have assumed that the total cost incurred is directly proportional to the travel time of vehicles, hence our objective is to minimize it. However, this is not always the case, as there are additional costs such as driver wages, toll charges, environmental impact fees and vehicle maintenance that can vary independently of travel time. On the other hand, the model treats waiting time in the same way as travelling time, which may not accurately reflect real-world cost structures where waiting incurs significantly lower costs.

Besides, the model assumes that all customers' time window, location and the demands at each node are predetermined and constant, which allows for a structured approach to route optimization. However, real-world logistics often involve dynamic variables that our model does not account for, such as last-minute changes in customer requests. For example, unexpected customer requests or cancellations could require on-the-fly adjustments to routes. Also, one limitation of using the Google Distance Matrix API for travel time estimation is that it requires the specification of a future date and time. While this can be useful for planning in advance, it assumes that traffic conditions and travel durations can be reliably predicted, which may not always align with real-time traffic patterns. Unforeseen delays due to accidents, weather, or sudden congestion cannot be accounted for in these estimates, potentially reducing the accuracy of the model under real-world conditions. Finally, for a general application, the number of nodes and available vehicles can be arbitrarily large, thus in the following subsections, we conduct an analysis on the model's computation time and explore methods to adapt our model for realistic parameters.

4.2. Computation Time Analysis

In this subsection, we conduct a computation time analysis for the algorithm proposed by running simulations across various node counts, n. The Python script utilised in this subsection can be found in **Appendix A.1.3**. The objective of this

analysis is to evaluate the model's efficiency and scalability as the problem size increases. The dataset for this analysis was generated using the following fixed parameters:

- $n_k = 3$: Number of vehicles
- $Q = 16$: Maximum capacity of each vehicle
- $T = 80$: Maximum allowable routing time

In addition to these fixed parameters, the following random values were used for each simulation instance:

- **Demands**: Randomly generated values between 1 and 3 for each customer node
- **Time Windows**: For each customer, random earliest arrival times were generated between 0 and 40, while the latest arrival times were set to 15 minutes after the earliest time.
- **Travel Times**: A random symmetric travel time matrix was generated, where travel times between nodes (excluding self-travel) ranged between 5 and 15 minutes.

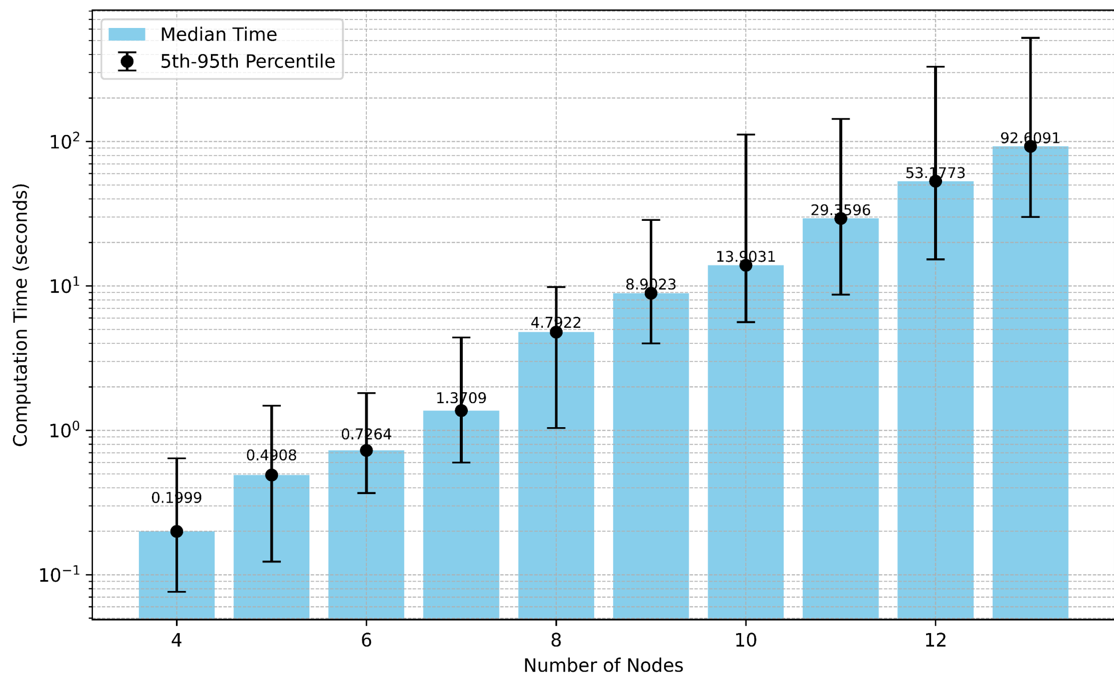

Figure 4. Computation times for node counts n ranging from 4 to 13, graphed on a logarithmic scale. The bars represent the median computation times obtained from 20 simulations for each node count, while the error bars indicate the 5th and 95th percentiles, representing the 90% confidence limits. The analysis was conducted using Google Colab.

The computation times were recorded and plotted for each node count n, ranging from 4 to 13, focusing on the median, 5th, and 95th percentiles, with each node count simulated 20 times, as illustrated in Figure 4. Given the relatively relaxed parameters of the constraints, all simulations returned feasible optimal solutions. The simulations were run in Google Colab, and it is worth noting that

these results can be significantly improved when run locally on a high-performance computer, as Colab's virtual environment imposes certain limitations on computational efficiency. The results project a clear trend: as the number of nodes increases, computation times grow significantly, with more variability at larger node counts. In particular, the problem becomes more computationally expensive beyond 10 nodes, with the upper 5 percent of simulations taking over a minute to compute.

This analysis highlights the importance of scalability for real-world applications, as the computation time increases exponentially with larger problem sizes. The data indicates that the model's performance is manageable for smaller node counts. Therefore, in the following subsection, we introduce an improvement to this model aimed at enhancing its efficiency for larger instances.

4.3. Scalability Using Node Partitions

In real-world scenarios, the number of nodes and available vehicles in a system can be exceedingly large. This increased complexity can lead to significant computational challenges, resulting in unrealistic computation times when solving for the optimal solution as demonstrated in Figure 4. To address this issue and improve scalability, we discuss an approach which splits the nodes into partitions, treating each partition as an independent sub-system. Each partition consists of a set of nodes, and vehicles are assigned from the fleet to service these nodes. The partitions can be based on user preferences or through clustering of nearby locations. This method provides several benefits:

- **Scalability**: It accommodates a larger number of nodes by breaking them into smaller sub-systems, allowing for a significant reduction in computational time.
- **Resource Optimization**: Vehicles can be allocated based on the specific demands and characteristics of each partition, leading to a more efficient use of resources.
- **Flexibility**: Departure stations and destinations can be set independently for each partition, enabling the service of customers to have a wider selection of destinations.

By implementing this partitioning method, organisations can enhance the effectiveness of the model proposed, leading to more efficient solutions within complex transportation networks.

4.4. Other Advanced VRP Methods

The selection of an appropriate VRP method depends on the specific features of the dataset, the constraints of the problem, and the desired balance between solution quality and computational time. In this subsection, we review existing available VRP tools and the work of other researchers, highlighting the unique use cases and applications for their VRP models.

4.4.1. Google OR-Tools

Google OR-Tools [7] is an open-source software suite designed for solving various

optimization problems, including the Vehicle Routing Problem (VRP), through advanced algorithms that enhance routing and scheduling efficiency. While OR-Tools is highly effective in delivering quick solutions for large datasets, our proposed model offers distinct advantages for small-scale logistics operations where specific operational constraints and passenger needs are prioritized. While Google OR-Tools is widely used for complex optimization problems across various industries, our model offers a simpler user interface that requires no technical expertise, making it more accessible for smaller organisations and non-technical users. This design allows operators to easily input parameters and constraints while delivering robust optimization solutions. It also incorporates a built-in troubleshooting system to easily identify infeasibilities due to parameter conflicts. Additionally, the organisations often benefit from focusing on constraint satisfaction, rather than only computational efficiency, which makes our approach valuable despite lacking the speed advantage of more generalized solvers.

4.4.2. AND/OR Precedence Constraints

The Vehicle Routing Problem with AND/OR Precedence Constraints and Time Windows framework proposed by Roohnavazfar *et al.* (2022) [4], addresses an additional constraint of requiring that for a given node, all AND-type predecessors must be visited before it and at least one OR-type predecessor must be visited before it. In the context of transport service, the AND precedence constraint becomes particularly relevant when there are multiple drop off locations in a single journey. To ensure all passengers are on board before commencing drop-offs, every drop-off node must be designated as a successor to all pickup nodes. This constraint guarantees that the vehicle only begins dropping off passengers after completing all pickups, avoiding contradicting routes, and ensuring a more organized and systematic journey for all passengers. Of course, this approach introduces significantly more computational complexities, as the number of constraints increases, making it more challenging to find optimal or even feasible solutions. Also, adhering to the precedence relationships may lead to less efficient routes in terms of distance or travel time.

4.4.3. Heuristic and Metaheuristic Algorithms

Heuristic algorithms [3] are approximation methods designed to find near-optimal solutions to complex optimization problems within an acceptable time-frame. There is no guarantee that these techniques will find an optimal solution but they are computationally efficient and are thus able to handle large data sets. Metaheuristics [4] represent a more sophisticated class of heuristic algorithms that incorporate mechanisms to explore the solution space more comprehensively. Some common metaheuristics for VRP include:

- **Local Search:** Local search algorithms [4] start with an initial solution and iteratively improve it by making small modifications within a defined neighbourhood. This process continues until no further improvements can be made within the current neighbourhood.

- **Simulated Annealing:** Drawing inspiration from the physical annealing process, simulated annealing [4] allows for occasional uphill moves, which are moves that temporarily worsen the solution. Accepting these less optimal moves helps the algorithm to break free from local optima. The probability of accepting such moves decreases as the search progresses, ensuring the algorithm eventually converges towards a high-quality solution.
- **Variable Neighbourhood Search (VNS):** VNS [5] utilizes a systematic approach to changing the neighbourhood structure during the search. It operates on the principle that a local optimum for one neighbourhood structure may not be a local optimum for another. By switching between various neighbourhoods, the algorithm aims to avoid getting stuck in local optima and explores a broader range of the solution space.
- **Large Neighbourhood Search (LNS):** LNS [5] works by removing a set of requests from the current solution and then reinserting them to create a new solution. The size of the neighbourhood is defined by the number of requests removed and reinserted. LNS is particularly well-suited for rich vehicle routing problems that incorporate various complexities, including time windows, heterogeneous fleets, and multi-dimensional capacity constraints.

5. Conclusions

In this project, we have proposed a complete elementary solution to the Vehicle Routing Problem (VRP). Unlike more advanced VRP solutions, which often require significant computational resources and specialized expertise, this model offers a practical balance of simplicity, cost-effectiveness, and adaptability for smaller-scale operations suitable for volunteer or charity organisations. The model's Python-based code is entirely written from scratch, making it straightforward to understand, troubleshoot, and modify. This eliminates the need to hire specialized technical personnel to implement or manage routing optimizations. One of the primary advantages of this model lies in its ease of use and self-sufficiency, as all concepts and parameters are comprehensively documented within this paper. This approach not only simplifies parameter input but also reduces the time and resources necessary for organisations to manage and adapt the model to their needs.

Additionally, the model can be extended to handle return journeys by inverting the start and end nodes, making it well-suited for transporting passengers both to and from their destinations. While computation time may be longer compared to advanced VRP methods, the trade-off is manageable for small-scale logistics organisations with smaller network of nodes and simpler routing requirements. Ultimately, this model is an elementary yet effective solution to the VRP, serving as a low-cost, efficient routing tool to provide essential transport services in a systematic, reliable, and economically feasible manner.

To build on this research, several extensions for future works could improve its practical applications. Automating the node partitioning process using an algorithm

to group nearby locations into subsystems could provide near-optimal solutions, allowing the script to handle larger problem instances. This approach would make the tool more robust for complex, real-world logistics by reducing problem complexity and focusing computational resources on smaller, manageable sub-problems. Such clustering strategies could significantly reduce computational expenses and time. Additionally, it is worth researching other heuristics or metaheuristic approaches to implement in the Python script, as these could further reduce the time required to generate near-optimal solutions compared to exact methods. These extensions would make the model better suited for real-time applications where quick, feasible approximations are more prioritized than exact solutions.

Acknowledgements

We would like to express our sincere gratitude to Jason Donaghy, manager of Fermanagh Community Transport, for his insights into real-world logistics operations and for providing crucial datasets that informed this work. Kai Lian is also thankful to Dr. Salissou Moutari, reader at Queen's University Belfast, whose feedback and assistance were invaluable in the completion of this research. The use of AI tools is acknowledged, which has facilitated a few aspects of this research, including assistance in scripting the Python code for solving the mathematical model. Finally, Kai Lian wishes to thank his colleague Jyoutir, whose inspiration led to the implementation of the Google Maps API, greatly enhancing the project's scope and applicability.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Transport: Fermanagh Community Transport: Northern Ireland. https://www.fermanaghcommunitytransport.com/

[2] Derbel, H., Jarboui, B., and Siarry, P. (2020) Green Transportation and New Advances in Vehicle Routing Problems. Springer International Publishing.

[3] Laporte, G. (2007) What You Should Know about the Vehicle Routing Problem. *Naval Research Logistics*, **54**, 811-819. https://doi.org/10.1002/nav.20261

[4] Roohnavazfar, M., Pasandideh, S.H.R. and Tadei, R. (2022) A Hybrid Algorithm for the Vehicle Routing Problem with and/or Precedence Constraints and Time Windows. *Computers & Operations Research*, **143**, Article 105766. https://doi.org/10.1016/j.cor.2022.105766

[5] Goel, A. and Gruhn, V. (2008) A General Vehicle Routing Problem. *European Journal of Operational Research*, **191**, 650-660. https://doi.org/10.1016/j.ejor.2006.12.065

[6] Google Distance Matrix API. https://developers.google.com/maps/documentation/distance-matrix/overview

[7] Google OR-Tools: Optimizing Routing and Scheduling Problems. https://developers.google.com/optimization

Appendix

In this section, we provide the complete Python script implemented for this proposed Vehicle Routing Problem (VRP).

A.1. Python Script

The complete Python code for solving the mathematical model, implementing Google API and conducting computation time analysis is presented. This Python script can be executed in any environment that supports Python, such as Jupyter Notebook or Google Colab. To ensure proper functionality, the following packages must be installed: Pulp, Requests, Matplotlib, and NetworkX. Use the following commands to install the required packages:

!pip install pulp
!pip install requests
!pip install matplotlib
!pip install networkx

A.1.1. Solution Using Pulp

The mathematical model proposed in Section 2 is solved using the following script, which generates the output shown in Figure 1. Users are required to input the parameters between lines 7 and 16. The travel time matrix, located at line 18, is automatically input from the Google Maps API request script, Listing A.1.2.

```
import pulp
import random

##################################################################################

# Parameters
n = 9  # Number of nodes
n_k = 3  # Number of vehicles
Q = 12  # Maximum capacity of each vehicle
M = 10000  # Arbitrary large integer
T = 90  # Maximum arrival time

demands = [0, 3, 2, 1, 2, 3, 2, 1, 0]  # Demands at each node

e = [0, 50, 40, 30, 20, 35, 45, 55, 0]  # Earliest arrival times
l = [M] + [i + 25 for i in e[1:-1]] + [M]  # Latest arrival times

travel_time = travel_time_matrix_minutes # Travel time matrix

##################################################################################

# Define the set
nodes = range(n)
vehicles = range(n_k)

# Initialize the problem
model = pulp.LpProblem("VRP", pulp.LpMinimize)

# Decision Variables
x = pulp.LpVariable.dicts("x", (nodes, nodes, vehicles), cat='Binary')
z = pulp.LpVariable.dicts("z", (nodes, vehicles), cat='Binary')
a = pulp.LpVariable.dicts("a", (nodes, vehicles), lowBound=0, cat='Continuous')
u = pulp.LpVariable.dicts("u", (vehicles), cat='Binary')

# Objective Variable
max_a_k = pulp.LpVariable.dicts("max_a", vehicles, lowBound=0, cat='Continuous')

# Objective Function
model += pulp.lpSum(max_a_k[k] for k in vehicles)

# Constraints

# 0. Initial conditions
```

```
44 for k in vehicles:
45     model += a[0][k] == 0  # Arrival time at depot is 0
46     model += z[0][k] == u[k]  # All vehicles depart from the depot
47     model += z[n-1][k] == u[k]  # All vehicles reach the drop
48 
49 # 1. Each node is visited exactly once by any vehicle
50 for i in nodes:
51     if i not in [0, n-1]:
52         model += pulp.lpSum(z[i][k] for k in vehicles) == 1
53 
54 # 2. Vehicle capacity constraint
55 for k in vehicles:
56     model += pulp.lpSum(demands[i] * z[i][k] for i in nodes) <= Q * u[k]
57 
58 # 3. Sequence logic and time flow
59 for i in nodes:
60     for j in nodes:
61         for k in vehicles:
62             model += 1 + M * (z[i][k] + z[j][k] - 2) <= x[i][j][k] + x[j][i][k]
63             model += a[j][k] + M * (1 - x[i][j][k]) >= a[i][k] + travel_time[i][j]
                   - M * (2 - z[i][k] - z[j][k])
64 
65 # 4. Time window constraints for each customer
66 for k in vehicles:
67     for i in nodes:
68         if i != 0:
69             model += e[i] * z[i][k] <= a[i][k]
70             model += a[i][k] <= l[i] * z[i][k]
71 
72 # 5. The drop off node is the final station
73 for k in vehicles:
74     for i in range(1, n-1):
75         model += a[n-1][k] >= a[i][k] - M * (1 - z[i][k])
76 
77 # 6. The arrival time at the final node must not exceed T
78 for k in vehicles:
79     model += T >= a[n-1][k]
80 
81 # 7. Setting objective variable
82 for k in vehicles:
83     model += max_a_k[k] == a[n-1][k]
84 
85 # Solve the problem
86 model.solve()
87 
88 # Troubleshooting System for Infeasibility
89 if pulp.LpStatus[model.status] != "Optimal":
90     print("Status:", pulp.LpStatus[model.status])
91 
92     capacity_violated = False
93     for k in vehicles:
94         total_demand = pulp.value(pulp.lpSum(demands[i] * z[i][k] for i in nodes))
95         if total_demand > Q:
96             capacity_violated = True
97             break
98 
99     # Print troubleshooting messages based on violations
100     if capacity_violated:
101         print("Vehicle capacity not satisfied.")
102     else:
103         print("Time window not satisfied.")
104 
105 # If feasible, print results with vehicle routes
106 else:
107     print("Total Routing Time = ", pulp.value(model.objective))
108 
109     vehicle_routes = {k: [] for k in vehicles}
110 
111     for k in vehicles:
112         visited_nodes_with_times = [
113             (i, a[i][k].varValue) for i in nodes if z[i][k].varValue > 0.5
114         ]
115         visited_nodes_with_times.sort(key=lambda x: x[1])
116         vehicle_routes[k] = visited_nodes_with_times
117 
118     # Display the route for each vehicle in a descriptive format
119     for k in vehicles:
120         if len(vehicle_routes[k]) > 1:
121             route_description = f"Vehicle {k} moves from "
122             prev_arrival_time = 0
123 
124             # Collect node details
125             node_descriptions = []
126             for idx, (node, arrival_time) in enumerate(vehicle_routes[k]):
127                 if idx > 0:
128                     previous_node = vehicle_routes[k][idx - 1][0]
129                     travel_time_value = travel_time[previous_node][node]
130                     waiting_time = arrival_time - (prev_arrival_time +
                        travel_time_value)
131                     actual_arrival_time = arrival_time - waiting_time
132                 else:
```

```
133                        waiting_time = 0
134                        actual_arrival_time = arrival_time
135
136                    prev_arrival_time = arrival_time
137                    node_descriptions.append(
138                        f"node {node} (arrives at time {actual_arrival_time:.2f},
                           waits for {waiting_time:.2f} min)"
139                    )
140
141                route_description += " -> ".join(node_descriptions)
142                print(route_description)
```

Listing 1. Vehicle routing problem optimisation script.

A.1.2. Google Maps API

This script must be executed first to generate the travel time matrix required in Listing A.1.1, as illustrated in Table 1. Users should input their API key on line 7, and the address of the nodes starting from line 10. It is recommended to include the zip code to accurately identify and differentiate places with generic names. Additionally, the number of days in advance for which travel data is needed should be specified on line 21, while the departure time should be entered on line 22. The script outputs two matrices: one for the travel time, used in Listing A.1.1, and another annotated matrix for further details and reference.

```
1  import requests
2  import time
3  from datetime import datetime, timedelta
4
5  ###############################################################################
6
7  api_key = '# API key'
8
9  nodes = [
10     'Ballinamallard, Enniskillen BT94 2FA',  # Node 0, Departure location
11     'Belleek, Enniskillen BT93 3FY',          # Node 1
12     'Brookeborough, Enniskillen BT94 4EY',    # Node 2
13     'Clabby, Fivemiletown BT75 0QZ',          # Node 3
14     'Derrygonnelly, Enniskillen BT93 6GA',    # Node 4
15     'Irvinestown Enniskillen BT94 1EN',       # Node 5
16     'Maguiresbridge, Enniskillen BT94 4RY',   # Node 6
17     'Lisnaskea, Enniskillen BT92 0FL',        # Node 7
18     'Fairgreen Shopping Centre, Forthill St, Enniskillen BT74 6JA'  # Node 8, Drop
           off location
19 ]
20
21 start_time = datetime.now() + timedelta(days=0) # Number of days ahead
22 start_clock = start_time.replace(hour=10, minute=15, second=0) # Departure time
23
24 ###############################################################################
25
26 def get_travel_time_row(api_key, origin, destinations, future_time):
27     url = 'https://maps.googleapis.com/maps/api/distancematrix/json?'
28
29     future_unix_timestamp = int(time.mktime(future_time.timetuple()))
30
31     params = {
32         'origins': origin,
33         'destinations': '|'.join(destinations),
34         'key': api_key,
35         'mode': 'driving',
36         'departure_time': future_unix_timestamp
37     }
38
39     response = requests.get(url, params=params)
40     data = response.json()
41
42     if response.status_code != 200:
43         raise Exception(f"Error: API request failed with status code {response.
               status_code}")
44
45     if data['status'] != 'OK':
46         error_message = data.get('error_message', 'No error message provided')
47         raise Exception(f"Error in API response: {data['status']} - {error_message
               }")
48
49     travel_time_row = []
50     for element in data['rows'][0]['elements']:
51         if element['status'] == 'OK':
52             travel_time = element['duration_in_traffic']['value']
53         else:
```

```
                travel_time = None
            travel_time_row.append(travel_time)

        return travel_time_row

def generate_travel_time_matrix(api_key, nodes, start_clock):
    travel_time_dict = {}
    travel_time_matrix = []

    for i, origin in enumerate(nodes):

        travel_times = get_travel_time_row(api_key, origin, nodes, start_clock)
        travel_time_matrix.append(travel_times)

        for j, travel_time in enumerate(travel_times):
            travel_time_dict[(i, j)] = round(travel_time / 60, 2) if travel_time
                is not None else None

        time.sleep(0.1)  # Add a delay to respect API rate limits

    return travel_time_dict, travel_time_matrix

# Generate travel time matrices
travel_time_dict, travel_time_matrix = generate_travel_time_matrix(api_key, nodes,
     start_clock)

# Convert matrix to minutes
travel_time_matrix_minutes = [[(round(time / 60, 2) if time is not None else None)
     for time in row] for row in travel_time_matrix]

# Print annotated travel time matrix
print(f"Travel Time Matrix (in minutes) for {start_clock.strftime('%Y-%m-%d %I:%M
    %p')}:")
print("\n\t" + "\t".join([f"Node {j}" for j in range(len(nodes))]))

for i, origin in enumerate(nodes):
    print(f"Node {i}:", end="\t")
    for j in range(len(nodes)):
        travel_time = travel_time_matrix_minutes[i][j]
        print(f"{travel_time:.2f}" if travel_time is not None else "N/A", end="\t"
            )
    print()

# Print simple matrix without annotations
print("\nSimple Travel Time Matrix (in minutes):")

for row in travel_time_matrix_minutes:
    print("\t".join([f"{time:.2f}" if time is not None else "N/A" for time in row
        ]))
```

Listing 2. Google map API request script.

A.1.3. Computation Time Analysis

This script is utilized in the Computation Time Analysis 4.2 to access the run time of the optimization script in Listing A.1.1. The model's fixed parameters are located between lines 10 and 13, while the range of the random parameters, which can be altered according to needs, are specified between lines 16 and 21. Additionally, the number of nodes and the number of simulations for each set of nodes are entered in lines 24 and 25. This script prints the 5th and 95th percentiles, along with the median of the time taken for the script to run for each node count. Additionally, it generates a histogram of the median on a logarithmic scale, with error bars representing the 5th and 95th percentiles. The number of feasible and infeasible states for each node count is also annotated below the histogram.

```
import pulp
import random
import time
import numpy as np
import matplotlib.pyplot as plt

##################################################################################

# Fixed parameters
n_k = 3       # Number of vehicles
Q = 16        # Maximum capacity of each vehicle
M = 10000     # Arbitrary large integer
T = 80        # Maximum arrival time

```

```
15 # Random parameters
16 def generate_random_parameters(n):
17     demands = [0] + [random.randint(1, 3) for _ in range(1, n - 1)] + [0]  #
           Demands range from 1 to 3
18     e = [0] + [random.randint(0, 40) for _ in range(1, n - 1)] + [0]  # Earliest
           windows range from 0 to 40
19     l = [M] + [(e[i] + 15) for i in range(1, n - 1)] + [M] # Latest windows are 15
            units after earliest
20     travel_time = [[0 if i == j else random.randint(5, 15) for j in range(n)] for
           i in range(n)] # Travel times are between 5 and 15
21     return demands, e, l, travel_time
22
23 # Simulation parameters
24 node_counts = [4, 5, 6]
25 simulations_per_node = 5
26
27 ##################################################################################
28
29 def solve_vrp(n, n_k, Q, M, T, demands, e, l, travel_time):
30     nodes = range(n)
31     vehicles = range(n_k)
32
33     model = pulp.LpProblem("VRP", pulp.LpMinimize)
34
35     x = pulp.LpVariable.dicts("x", (nodes, nodes, vehicles), cat='Binary')
36     z = pulp.LpVariable.dicts("z", (nodes, vehicles), cat='Binary')
37     a = pulp.LpVariable.dicts("a", (nodes, vehicles), lowBound=0, cat='Continuous'
           )  # Arrival times
38     u = pulp.LpVariable.dicts("u", (vehicles), cat='Binary')  # Vehicle
           utilization binary
39
40     max_a_k = pulp.LpVariable.dicts("max_a", vehicles, lowBound=0, cat='Continuous
           ')
41
42     model += pulp.lpSum(max_a_k[k] for k in vehicles)
43
44     for k in vehicles:
45         model += a[0][k] == 0
46         model += z[0][k] == u[k]
47         model += z[n-1][k] == u[k]
48
49     for i in nodes:
50         if i not in [0, n-1]:
51             model += pulp.lpSum(z[i][k] for k in vehicles) == 1
52
53     for k in vehicles:
54         model += pulp.lpSum(demands[i] * z[i][k] for i in nodes) <= Q * u[k]
55
56     for i in nodes:
57         for j in nodes:
58             for k in vehicles:
59                 model += 1 + M * (z[i][k] + z[j][k] - 2) <= x[i][j][k] + x[j][i][k
                       ]
60                 model += a[j][k] + M * (1 - x[i][j][k]) >= a[i][k] + travel_time[i
                       ][j] - M * (2 - z[i][k] - z[j][k])
61
62     for k in vehicles:
63         for i in nodes:
64             if i != 0:
65                 model += e[i] * z[i][k] <= a[i][k]
66                 model += a[i][k] <= l[i] * z[i][k]
67
68     for k in vehicles:
69         for i in range(1, n-1):
70             model += a[n-1][k] >= a[i][k] - M * (1 - z[i][k])
71
72     for k in vehicles:
73         model += T >= a[n-1][k]
74
75     for k in vehicles:
76         model += max_a_k[k] == a[n-1][k]
77
78     start_time = time.time()
79     model.solve()
80     computation_time = time.time() - start_time
81
82     return pulp.LpStatus[model.status], pulp.value(model.objective),
           computation_time
83
84 results = {}
85
86 for n in node_counts:
87     computation_times = []
88     feasible_count = 0
89     infeasible_count = 0
90
91     for _ in range(simulations_per_node):
92
93         demands, e, l, travel_time = generate_random_parameters(n)
94         status, optimal_cost, computation_time = solve_vrp(n, n_k, Q, M, T,
               demands, e, l, travel_time)
```

```
95
96          if status == "Optimal":
97              computation_times.append(computation_time)
98              feasible_count += 1
99          else:
100             infeasible_count += 1
101
102     median_time = np.median(computation_times) if computation_times else 0
103     percentile_5th = np.percentile(computation_times, 5) if computation_times else
            0
104     percentile_95th = np.percentile(computation_times, 95) if computation_times
            else 0
105
106     results[n] = {
107         'median': median_time,
108         '5th_percentile': percentile_5th,
109         '95th_percentile': percentile_95th,
110         'feasible_count': feasible_count,
111         'infeasible_count': infeasible_count
112     }
113
114     print(f"Node Count: {n}, Median Time: {median_time:.4f}, 5th Percentile: {
            percentile_5th:.4f}, 95th Percentile: {percentile_95th:.4f}")
115
116 labels = list(results.keys())
117 medians = [results[n]['median'] for n in labels]
118 percentile_5th = [results[n]['5th_percentile'] for n in labels]
119 percentile_95th = [results[n]['95th_percentile'] for n in labels]
120 feasible_counts = [results[n]['feasible_count'] for n in labels]
121 infeasible_counts = [results[n]['infeasible_count'] for n in labels]
122
123 x = np.arange(len(labels))
124 width = 0.4
125 fig, ax = plt.subplots(figsize=(10, 6))
126 rects = ax.bar(x, medians, width, label='Median Time', color='blue')
127
128 ax.errorbar(x, medians,
129             yerr=[medians - np.array(percentile_5th), np.array(percentile_95th) -
                    medians],
130             fmt='none',
131             ecolor='black',
132             capsize=5,
133             label='5th and 95th Percentile')
134
135 ax.set_ylabel('Computation Time (seconds)')
136 ax.set_xlabel('Number of Nodes, n')
137 ax.set_title('Computation Time for Feasible Solutions')
138 ax.set_xticks(x)
139 ax.set_xticklabels(labels)
140 ax.set_yscale('log')
141 ax.legend()
142
143 for i, (feasible, infeasible) in enumerate(zip(feasible_counts, infeasible_counts)
        ):
144     ax.text(i, 0.05, f'Feasible: {feasible}\nInfeasible: {infeasible}',
145             ha='center', va='top', fontsize=10, color='black', weight='bold')
146
147 plt.grid()
148 plt.tight_layout()
149 plt.ylim(bottom=0.1)
150 plt.show()
```

Listing 3. Computation time analysis script.