A web application solver in Python for the Vehicle Routing Problem with Time Windows (VRPTW)

Ana Carolina Bergamasco Perez[a], Alexandre Checoli Choueiri[1]

[a]*Industrial Engineering, Universidade Federal do Paraná - UFPR*

Abstract

This study presents a web application developed in Python to solve the Vehicle Routing Problem with Time Windows (VRPTW). The solver employs a greedy algorithm to generate the solution, and based on the user's uploaded customer list and vehicle fleet details, it provides the route planning for the set of customers. The system utilizes external APIs for geolocation and distance matrix calculations, ensuring accurate and efficient routing. Developed to be accessible and democratize vehicle routing optimization solutions for medium and small organizations, the tool offers satisfactory results for real-life datasets, helping to enhance efficiency and reduce logistics-related operational costs.

Keywords:
VRPTW, Python, Operational research, API

1. Introduction

The post-pandemic logistic industry has undergone significant changes driven by digitization, automation, sustainability and the growth of e-commerce business. The field that studies all the activities related to freight movement in a city is called Urban Logistics, which involves the analysis, planning, maintenance, and improvement of logistics activities [1]. Technology is an important fuel for those improvements, as it increases competitive advantage, enabling cost reduction in logistics operations.

In a large city, one of the main challenges for Urban logistics is the route planning of delivery/pickup services in a way that minimizes transportation costs and the time spent in the process. The Vehicle Routing Prob-

lem (VRP) is a problem in operational research that searches for a solution to this challenge.
The VRP consists in defining routes for a fleet of vehicles departing from a central warehouse (depot) to serve a set of clients. One extension for this problem is the Vehicle Routing Problem with Time Windows (VRPTW), which includes time windows as constraints for client service. Thus, the vehicle can only collect/deliver the product within the time interval established by the client.
This paper aims to develop an open-source web application to provide a solution for the VRPTW. The coding language used was Python, that can be understood and modified by medium-level programmers. That way, medium and small organizations can also have access to an optimization tool, to leverage their business and improve their service. The users of the tool can upload their clients' list and receive as an output a set of routes that serve all clients.
The remainder of this paper is organized as follows: a literature review on the VRP is provided in Section 2. Section 3 describes the algorithm used to optimize the VRPTW, as well as the necessary APIs used to extract external data, and how they all connect to each other. Finally, in Section 4 we show the software UI and the conclusion is presented in 5.

2. Literature review

As previously mentioned, the VRP is an optimization problem that aims to reduce transportation costs and time spent in logistic operations by defining optimal routes to serve a set of clients [2]. It was introduced as a generalized problem of the Travelling Salesman Problem (TPS) in 1959 by Dantzig and Ramser [3] [4], and it has been wildly studied and applied since then, not only in theory, but also in real scenarios. One of its main variants is known as VRPTW, which incorporates time intervals to the set of hard constraints. It can be defined as follow:
Let $G = (V, A)$ be a graph where $V = 1, \ldots, n$ is a set of vertices representing geographically dispersed clients, where V1 is the depot, and A is a set of arcs. For every arc (i, j) where $i <> j$ there is a cost C_{ij} associated. This cost can be interpreted as the transportation cost, but also as the travel time (t_{ij}) between clients. The service time for each client is also included in t_{ij}. Each client is visited only once, and the vehicle route must start and end at the depot (Figure 1 displays a solution for

a VRP). For the VRPTW, there is a time window assigned to each client, and if the vehicle arrives before the time window opens, it needs to wait to start the service. If it arrives when the time window has already closed, then the solution becomes unfeasible. The objective of the problem, in most cases, is to minimize the number of routes used, the time spent, or the distance traveled to serve all clients, while adhering to the constraints established during the problem formulation.

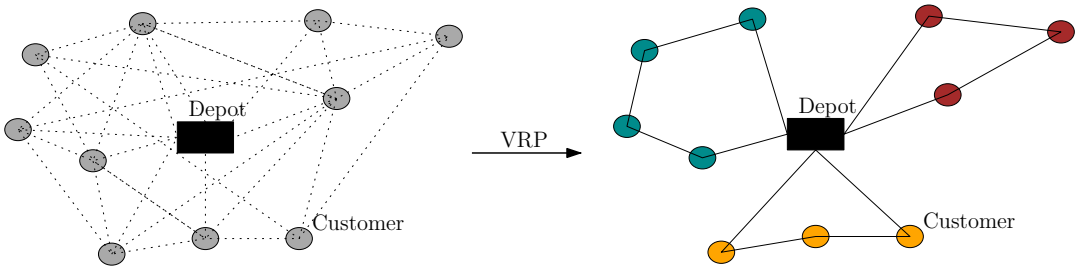

Figure 1: Feasible solution for the VRP

The VRP and its variants are considered to be an NP-hard problem due to its complexity, and although there are exact algorithms that provide an optimal solution, like the Branch and Bound method utilized by [5] or the column generation method proposed in [6] , the computational cost required is high and the solution is limited to simpler problems [2]. As a result, researchers prefer to study the application of heuristic methods, which, although there is no guarantee that an optimal solution will be reached, present satisfactory results for medium and large complexity problems at a much lower computational cost.

In the literature, many heuristic and metaheuristic methods for solving the Vehicle Routing Problem with Time Windows (VRPTW) can be found, as per example: Tabu Search (TS) [7], Ant Colonization Optimization (ACO) and variants, such as Hybrid Ant Colonization Optimization (HACO) [8], Genetic Algorithm (GA), Harmony Search Algorithm (HSA) hybridized with local search algorithms (such as Simulated Annealing-SA, Great Deluge-GD, and Hill Climbing-HC) [9], Particle Swarm Optmization (PSO) [10], among various others.

In [11], the authors introduced a variable iterated greedy algorithm to solve the Traveling Salesman Problem with time windows (TSPTW). The greedy algorithm is hybridize with a Variable Neighborhood Search Algorithm (VNS), which improves solution quality by altering neighboring

solutions through the destruction and reconstruction of solution components, employing 1-Opt local searches. The results obtained demonstrate that the hybrid greedy algorithm performs competitively in solving route planning-related problems compared to other existing algorithms.
In this context, various solutions to the VRPTW are documented in the literature, implementing diverse algorithms to achieve the most advantageous outcomes. The relevance of these solutions is underscored by the numerous vehicle routing optimization challenges in the logistics and transportation sectors, particularly in operations within large cities [12].
Companies can benefit from employing vehicle routing problem solvers in last-mile deliveries [12][2]. Despite the availability of numerous VRP-solving software solutions on the market, they are often not accessible to smaller companies and individual entrepreneurs. This situation underscores the need for developing affordable tools, making it a relevant area of study for the sector. Open-source applications are particularly useful in broadening the accessibility of VRPTW solvers [2]. Python is especially advantageous for creating such products due to its versatility, readability, and extensive ecosystem of libraries and frameworks. Furthermore, Python's compatibility with multiple platforms and its strong support for integration with other technologies make it ideal for building cross-platform open-source applications.

3. Development

This section describes the main components of the project and how they work. To facilitate the structuring and organization of the project, the components were grouped into three different modules: the Python script, the user interface (Streamlit), and the external connections (APIs). The software architecture in Figure 2 defines the overall organization of the system, including its main components and the interactions between them. This approach makes it easier to implement changes without negatively impacting the development of the project.

3.1. Greedy algorithm

The present paper utilizes the greedy algorithm, a constructive heuristic, to generate an outcome for the VRPTW. It is known as "constructive" because it starts from an empty solution, building the solution in

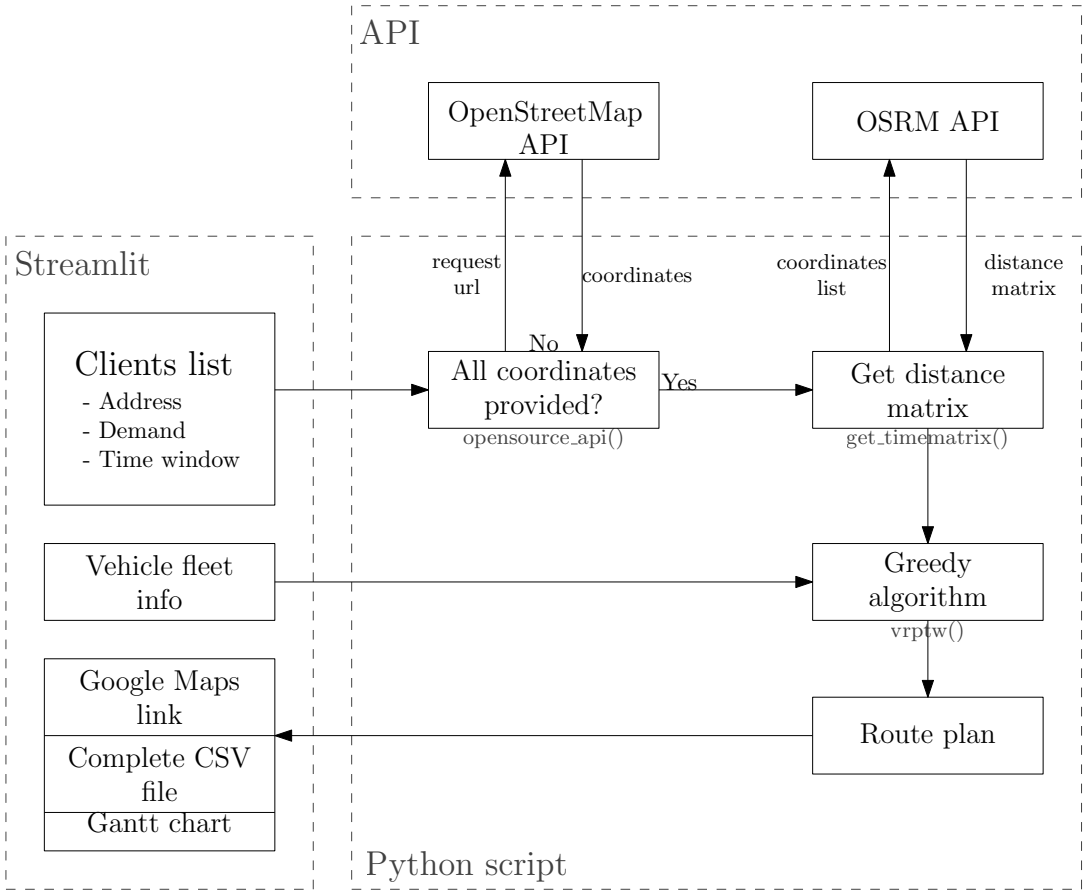

Figure 2: Simplified software architecture

parts, with each iteration, and as “greedy” because the decision in each iteration is based on the most advantageous available choice.
This algorithm is usually used to generate an initial solution for optimization problems, and then other optimization methods are used to improve the existing solution.
For this study, the greedy algorithm was fed with a list of customers containing the demand, the time windows and the service time for each client, the distance matrix, and the vehicle capacity. Based on those details, in each iteration, the algorithm selects the client whose time window closes the earliest, adhering to the vehicle’s capacity and time window constraints.
The execution of the algorithm is defined using two *while* loop structures: one that ensures all clients on the list are visited, and the other

Algorithm 1: Greedy algorithm for the VRPTW

Input : Clients list C with the demand $d[i]$ and timewindows $[s[i], e[i]]$ for each client, distance matrix $T[i][j]$, service time $m[i]$, and vehicle capacity Q

Output: Set of routes R

$R \leftarrow \emptyset$;
$clients \leftarrow$ Earliest(C) ; `// Sort the clients list based on timewindow closing time`
$clients \leftarrow \{clients \setminus clients[0]\}$;
while length$(clients) \neq 0$ **do**
  $route \leftarrow C[0]$; `// The route starts from the depot`
  $end\_service \leftarrow C[s][0]$; `// The vehicle leaves the depot the moment it opens`
  $load \leftarrow 0$;
  $j \leftarrow 0$;
  $i \leftarrow 0$;
  while $(load < Q) and (i <= length(clients))$ **do**
    if $(C[d][clients[i]] + load \leq Q)$ and $(end\_service + T[route[j]][clients[i]] \leq C[e][clients[i]])$ **then**
      $route \leftarrow route + clients[i]$;
      $load \leftarrow load + C[d][clients[i]]$;
      if $(end\_service + T[route[j]][clients[i]] \leq C[s][[clients[i]])$ **then**
        $end\_service \leftarrow C[s][clients[i]] + C[m][clients[i]]$;
      else
        $end\_service \leftarrow end\_service + T[route[j]][clients[i]] + C[m][clients[i]]$;
      end
      $clients \leftarrow \{clients \setminus clients[i]\}$;
      $j \leftarrow j + 1$;
      $i \leftarrow 0$;
    else
      $i \leftarrow i + 1$;
    end
  end
  $route \leftarrow route + C[0]$; `// The vehicle must return to the depot`
  $R \leftarrow R \cup route$;
end
return R;

one for determining the routes, as in algorithm number 1.
Initially, the clients list is sorted based on the end time of the time window. This way, the list is organized from the customer whose time window closes earliest to the customer whose time window closes latest. Then, the first *while* loop initiates, and the route commences with client 0, representing the depot (the vehicle always starts the route from the depot and concludes by returning to it). Subsequently, the *while* loop in line 10 along with the *if* statement in line 11 start to increase the route, adding more customers to it. The criteria for adding a client to the route are as follows:

- The client has not been visited previously;
- The client's demand, when combined with the current load in the vehicle, does not exceed the vehicle's capacity;
- The vehicle's arrival time at the client (defined in line 11 as the sum between the time that the vehicle left the previous customer and the travel duration between the previous customer and the next one) must be within their respective time window.

If all conditions are met, the client is incorporated into the route. Upon selecting the next client, it is essential to update the vehicle's available capacity and the timestamps associated with the service at the client's location, to ensure that the vehicle can serve all clients on the route while adhering to all time windows.
There are three key timestamps for the service: the arrival time of the vehicle at the client's location, the moment when the service begins, and the moment it ends (the moment the vehicle leaves the location — this one being the most important). The arrival time is determined by adding the departure time from the previous client to the travel duration between the previous client and the next one. The moment the service starts depends on whether the vehicle arrives within the designated time window or before it opens. If the arrival time falls within the time window, the service can start immediately (arrival time = service start time) (see line 17 of algorithm 1); otherwise, the service starts the moment the time window opens (see line 15 of algorithm 1). The end of the service is defined by adding the duration of the service to the start time.

Once the timestamps are calculated, the newly incorporated client is removed from the clients list, and the variables i and j are updated (lines 20 and 21 of algorithm 1). The variable i is utilized to iterate through the clients list, whereas the variable j is utilized to iterate through the current route.
When no further candidates are available to form a route (when one of the aforementioned constraints is not met), the current route is finalized by returning the vehicle to the depot and is appended to the set of generated routes. If clients remain to be visited, a new route is initiated, provided there are available vehicles to undertake the journey. Once all clients have been visited, the stop condition of the *while* loop is satisfied, terminating the execution of the greedy algorithm.

3.2. APIs

To begin solving the VRPTW, the user must upload a list of clients (in a CSV file). Initially, the list must contain the address, demand, and service's time window for each client. From the uploaded list, there are two paths that can be taken, depending on whether or not the coordinates (latitude and longitude) of each address were provided. Once the code starts, it checks if all coordinates have been provided. If not, the opensource_api() function is called. After the coordinates are retrieved, the code uses the get_timematrix() function to generate the distance matrix between each customer in order to start the greedy algorithm solution. Both the opensource_api() function and the get_timematrix() function request data from an API.
An API (Application Programming Interface) is an interface that allows systems to communicate (share data) through a "requests and response" dynamic. APIs are widely used in software development, as they help to simplify and speed up the development process, allowing existing solutions to be used in the creation of a new application.

3.2.1. opensource_api() function

One of the APIs utilized for the development of this work is the OpenStreetMap API, an open-source API that provides map data, allowing for location searches based on addresses.
The function opensource_api() makes requests to this Search API in order to obtain the latitude and longitude of the addresses provided by the user. The following request format was used to obtain the coordinates:

https://nominatim.openstreetmap.org/search?<params>&format=jsonv2

A request is made for each address in the list. The API returns a response in JSON format (JavaScript Object Notation), a lightweight format for data interchange. The code uses the *json* library to decode the data structure into a dictionary, making it easier to extract the data of interest (latitude and longitude). Using the address "RUA LAUDELINO FERREIRA LOPES,229,NOVO MUNDO,Curitiba,Parana,Brasil" as an example, the request URL and the JSON output are as follows:

https://nominatim.openstreetmap.org/search?q=RUA LAUDELINO FERREIRA LOPES,229,NOVO MUNDO,Curitiba,Parana,Brasil&format=jsonv2

After decoding the JSON output as in Figure 3, into a Python dictionary, it is easy to access and extract the latitude ("lat") and longitude ("lon") data. The opensource_api() function returns a list containing the coordinates in the format required to be used in the get_timematrix() function, as well as inserts the coordinate data into the spreadsheet provided by the user.

3.2.2. get_timematrix() function

This function utilizes the OSMR (Open Source Routing Machine) API to generate the distance matrix that will be used in the greedy algorithm solution. This API offers different services related to route planning. Among them, the Table service was chosen for the application, as it provides the duration and/or distance of the fastest route between two pairs of coordinates. The request was made as follow:

http://router.project-osrm.org/table/v1/driving/<coordinates>?sources=str(j)

Initially, a for loop is used to set up the link for the request, using the list of coordinates (in string format) generated in the opensource_api() function. The <coordinates> parameter is filled in with all the coordinate pairs. Given 3 pairs of coordinates: [-49.2882727,-25.4589783], [-49.3045595,-25.5068925], and [-49.2234297,-25.4766991], the URL request is set as follows:

```
[
  {
    "place_id": 38809267,
    "licence": "Data OpenStreetMap contributors, ODbL 1.0.
       http://osm.org/copyright",
    "osm_type": "way",
    "osm_id": 33068435,
    "lat": "-25.5068925",
    "lon": "-49.3045595",
    "category": "highway",
    "type": "tertiary",
    "place_rank": 26,
    "importance": 0.10001,
    "addresstype": "road",
    "name": "Rua Laudelino Ferreira Lopes",
    "display_name": "Rua Laudelino Ferreira Lopes, Capao
       Raso, Curitiba, Regiao Geografica Imediata de
       Curitiba, Regiao Metropolitana de Curitiba, Regiao
       Geografica Intermediaria de Curitiba, Parana, Regiao
       Sul, 81150-120, Brasil",
    "boundingbox":["-25.5204931","-25.4926184","-49.3051507"
       ,"-49.3043058"]
  }
]
```


Figure 3: OpenStreetMap API JSON output

https://router.project-osrm.org/table/v1/driving/-49.2882727,-25.4589783;-49.3045595,-25.5068925;-49.2234297,-25.4766991?sources=0

The API has a limit of 512 requests per connection. Therefore, for lists with numerous clients, it would not be possible to generate the entire matrix with just one request. This problem was solved by making a request for each client. The <source> parameter is used to indicate which pair of coordinates is used as the reference point to calculate the distance. The variable j traverses the list of coordinates (j=0,..., len(coordinates)), being incremented with each iteration. Thus, in each

```
1  {
2    "code": "Ok",
3    "destinations": [
4      {
5        "distance": 14.470048621,
6        "name": "Rua Murilo do Amaral Ferreira",
7        "location": [-49.288259, -25.458848]
8      },
9      {
10       "distance": 0,
11       "name": "Rua Desembargador Ernani Guarita Cartaxo /
             Rua Laudelino Ferreira Lopes",
12       "location": [-49.30456, -25.506893]
13     },
14     {
15       "distance": 9.182194123,
16       "name": "Rua Araticum",
17       "location": [-49.22334, -25.476685]
18     }
19   ],
20   "durations": [[0, 684.3, 822.9]],
21   "sources": [
22     {
23       "distance": 14.470048621,
24       "name": "Rua Murilo do Amaral Ferreira",
25       "location": [-49.288259, -25.458848]
26     }
27   ]
28 }
```

Figure 4: OSRM API JSON output

request, the distance of a pair of coordinates relative to all others is requested (1-to-many dynamic). In the example above, the 'source' parameter is set '0', i.e., the API will return the duration, in seconds, to reach coordinates 2 and 3, starting from the first coordinate.
This API also returns a JSON format response, as in Figure 4, that contains, in line 20, a 1 x n duration matrix (where n is the number of

clients). After the decoding, the duration matrix is retrieved as a list. When all positions in the coordinates list have been traversed, the duration lists are appended into a dataframe, thus forming the distance matrix.

3.3. Streamlit

To facilitate communication between the Python script and the user, an interface was created using the *Streamlit* library. Streamlit is a package that transforms Python scripts into websites without the need to use front-end languages.
The developed interface presented in Figure 5 allows the user to input information related to their business, such as their customer list and vehicle fleet.

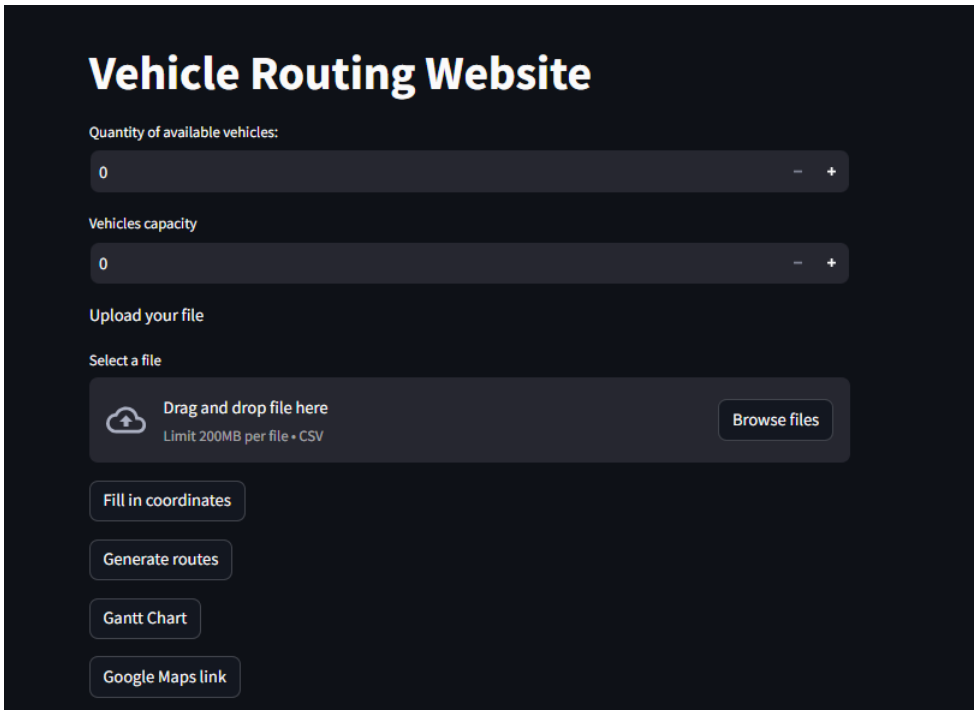

Figure 5: Initializing the interface

Once the necessary information is entered, the user can generate routes to serve their set of customers. Additionally, it is possible to download a spreadsheet containing information on latitude and longitude, routes, and the start and end times of service. To offer a more interactive and

user-friendly visualization of the results, a Google Maps link with the route and a Gantt chart, which shows which customers are visited on each route and at what times the service is performed, are also generated.
One important point to highlight is the use of the *session.state* feature, which allows data to persist across reruns of the application. Every time a button is pressed, Streamlit reruns the entire Python script, causing the state of the utilized variables to reset. To circumvent this issue, variables used by more than one object and that needed to maintain their state after a rerun (variables entered by the user) were stored in session.state.

4. Results

In this section, the results of the VRPTW solver implemented in Python are presented. To evaluate the performance of the interface and the effectiveness of the developed greedy algorithm, tests were conducted using lists of customers of varying sizes. The experiments were executed on a laptop computer with the following specifications: an Intel i3 CPU running at 2 GHz and 12 GB of RAM. This configuration represents a modest processing capacity and reflects the computers that would be used in practice, ensuring that the implementation was evaluated in a typical development environment.
Due to the necessity of interacting with external APIs, an internet connection was essential for proper request functionality. This allowed for real-time data acquisition and the use of external services crucial for route validation and geographic data collection from customers.
For benchmark, a real world VRP instance was utilized for the tests, and four datasets were extracted from it, containing 10, 30, 70, and 100 customer addresses, respectively. The time windows and demands of each customer were randomly generated within predefined intervals. The time windows ranged from 8:00 to 17:00, corresponding to the warehouse's operating hours, while the demands were allocated within the range of 1 to 20 units. Table 1 presents the execution times for each dataset.
For each test , the fleet size was defined to match the number of customers in the list, ensuring that the number of vehicles available was sufficient to serve all customers and that the problem's solution would be feasible. The vehicle capacity was kept constant across all scenar-

Instances			VRPTW Solver	
Number of clients	Fleet size	Vehicle capacity	Generated routes	Execution time (seconds)
10	10	45	4	40
30	30	45	8	180
70	70	45	17	720
100	100	45	26	1,320

Table 1: Computational results on benchmark instances

Client	Name	Address	Number	Neighborhood	City	State
0	ADEMIR JOSÉ VIEIRA	RUA LAUDELINO FERREIRA LOPES	229	NOVO MUNDO	Curitiba	Parana
8	ALIATAR SILVA NETO	RUA JÚLIA WANDERLEY	57	MERCÊS	Curitiba	Parana
1	ADRIANE ANGERER ULIANA	RUA MURILO DO AMARAL FERREIRA	72	ÁGUA VERDE	Curitiba	Parana
6	ALEXANDRE SHIGUERU ARAKI	AVENIDA SILVA JARDIM	1364	REBOUÇAS	Curitiba	Parana
9	ÁLVARO FONSECA KAMINSKI	AVENIDA REPÚBLICA ARGENTINA	357	ÁGUA VERDE	Curitiba	Parana
7	ALEXANDRINA ZAPOCTOCZNY BERTAPELI	RUA ANTÔNIO TURÍBIO TEIXEIRA BRAGA	280	BUTIATUVINHA	Curitiba	Parana
2	ALESSANDRO DA SILVA	RUA ARATICUM	214	UBERABA	Curitiba	Parana
3	ALEXANDRE ALMEIDA BLITZKOW	RUA ALFERES ÂNGELO SAMPAIO	1495	BATEL	Curitiba	Parana
4	ALEXANDRE AUGUSTO LEAL	RUA JOÃO GUARIZA	522	SÃO LOURENÇO	Curitiba	Parana
5	ALEXANDRE FRIEDRICH ALLAGE	RUA ENGENHEIRO IVAN RIGOMERO CECCON	239	JARDIM SOCIAL	Curitiba	Parana

Table 2: Output table Part 1

ios, allowing for fair and consistent comparisons between different customer list sizes.

The execution times in Table 1 indicates that the solver is capable of planning routes efficiently for up to 100 customers within a reasonable time frame. Additionally, the tests were conducted with pre-filled coordinates in the spreadsheet to reduce processing time. The coordinates for the 100 addresses were obtained through the solver prior to testing, with a processing time of 100 seconds. This additional time for obtaining the coordinates did not significantly impact overall performance, highlighting the solver's efficiency in both generating the coordinates and planning the subsequent routes.

4.1. Interface functionalities

The developed interface offers different functionalities that facilitate the use of the algorithm and the analysis of the results:

- Spreadsheet Download: Users can download a spreadsheet containing the coordinates of each address. Additionally, a final spreadsheet can be generated, showing the routes and the timestamps of service for each customer, as in Tables 2 and 3.

Country	Lat	Long	S. t-window	E. t-window	Service time	Demand	Arrival	S. service	E. service
Brasil	-25.5068925	-49.3045595	08:00	17:00		0			
Brazil	-25.422665	-49.2853795	08:00	08:57	600	18	08:17:00	08:17:00	08:27:00
Brazil	-25.4589783	-49.2882727	13:28	13:47	600	18	08:36:00	13:28:00	13:38:00
Brazil	-25.4416931	-49.2741144423068	14:25	14:44	600	18	08:14:00	14:25:00	14:35:00
Brazil	-25.4495105	-49.2878217	14:30	15:07	600	12	14:38:00	14:38:00	14:48:00
Brazil	-25.4030067	-49.3476178	14:12	16:10	600	12	15:03:00	15:03:00	15:13:00
Brazil	-25.4766991	-49.2234297	12:35	16:04	600	19	08:18:00	12:35:00	12:45:00
Brazil	-25.4404982	-49.2843453	15:46	16:35	600	18	12:58:00	15:46:00	15:56:00
Brazil	-25.3911545	-49.2657273	16:08	16:38	600	11	08:24:00	16:08:00	16:18:00
Brazil	-25.4128549	-49.2354046	16:11	16:56	600	17	16:25:00	16:25:00	16:35:00

Table 3: Output table part 2

- Graphical Visualization: For more intuitive results, the interface allows the generation of Gantt charts as in Figure 6, that display the customers in each route and the service times. This visualization aids in analyzing the temporal distribution of service.

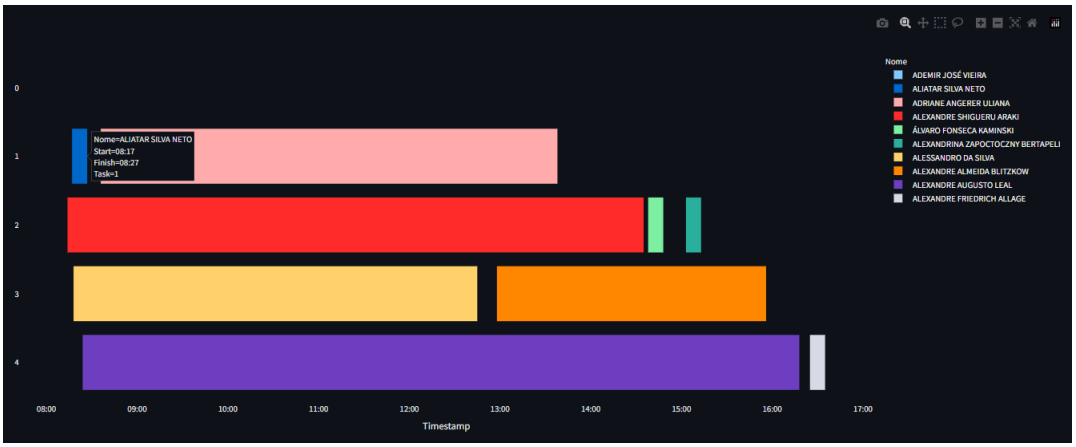

Figure 6: Gantt chart (routes x time)

- Google Maps Integration: An URL to Google Maps is generated, enabling the geo-spatial visualization of the planned routes, as in Figure 7, which allows a practical and visual analysis of the solver's output.

5. Conclusions

In this study, we presented the development and evaluation of a solver in Python for the Vehicle routing problem with time windows. The chosen algorithm to generate the routes was the Greedy algorithm, consid-

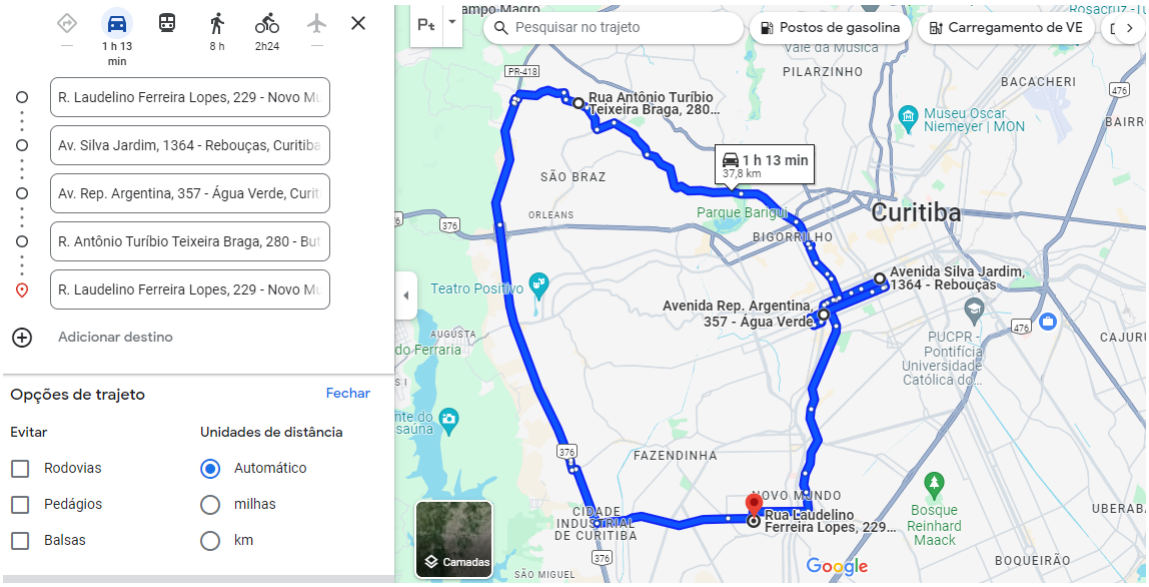

Figure 7: Map visualization for one of the routes

ering its ease of implementation and low computational cost. Additionally, the Python programming language was chosen for development due to its clear and intuitive syntax and its support in different platforms, which is particularly beneficial for open source projects. Furthermore, Python offers a vast collection of libraries and modules that can be re-utilized in new projects.

The application was tested using datasets of different sizes, and its performance was measured in terms of execution time. The results demonstrated the solver's ability to effectively generate feasible routes within a reasonable time frame, even for larger datasets.

The developed interface enhances the practical utility of the algorithm by offering various functionalities such as the ability to download detailed spreadsheets with customer coordinates, route information, and service timestamps, as well as an user-friendly visualization of the results with Gantt charts and routes map. These make the tool accessible and useful for practical applications.

To mention improving points, there are several areas for future work in the algorithm and the interface. Implementing post-optimization methods is a key next step to improve the quality of the routing solutions, potentially reducing costs and travel times. Additionally, integrating

more comprehensive APIs will enhance the accuracy and efficiency of the system by providing access to more extensive data sets. Developing an internal database to store previously used addresses will streamline future queries and improve the algorithm's efficiency.
To broaden the accessibility and usability of the application, there are plans to develop the code on an online code hosting platform such as GitHub. This will allow a wider community of users to benefit from the tool and contribute to its ongoing development and improvement.
In summary, the combination of the greedy algorithm with a user-centric interface has shown to be effective for solving routing problems in a practical setting, offering a significantly better approach than manually planning the routes. The proposed future directions aim to further enhance the functionality and efficiency of the tool, making it a robust solution for real-world applications in logistics and transportation management.

References

[1] Gülçin Büyüközkan and Öykü Ilıcak. Smart urban logistics: Literature review and future directions. Socio-Economic Planning Sciences, 81:101197, 2022.

[2] Güneş Erdoğan. An open source spreadsheet solver for vehicle routing problems. Computers & operations research, 84:62–72, 2017.

[3] George B Dantzig and John H Ramser. The truck dispatching problem. Management science, 6(1):80–91, 1959.

[4] Mourad Zirour. Vehicle routing problem: models and solutions. Journal of Quality Measurement and Analysis JQMA, 4(1):205–218, 2008.

[5] Antoon WJ Kolen, AHG Rinnooy Kan, and Harry WJM Trienekens. Vehicle routing with time windows. Operations Research, 35(2):266–273, 1987.

[6] Jacques Desrosiers, François Soumis, and Martin Desrochers. Routing with time windows by column generation. Networks, 14(4):545–565, 1984.

[7] Abdel-Rahman Hedar and Mohammed Abdallah Bakr. Three strategies tabu search for vehicle routing problem with time windows. Computer Science and Information Technology, 2(2):108–119, 2014.

[8] Qiulei Ding, Xiangpei Hu, Lijun Sun, and Yunzeng Wang. An improved ant colony optimization and its application to vehicle routing problem with time windows. Neurocomputing, 98:101–107, 2012.

[9] Esam Taha Yassen, Masri Ayob, Mohd Zakree Ahmad Nazri, and Nasser R Sabar. An adaptive hybrid algorithm for vehicle routing problems with time windows. Computers & Industrial Engineering, 113:382–391, 2017.

[10] Qichao Wu, Xuewen Xia, Haojie Song, Hui Zeng, Xing Xu, Yinglong Zhang, Fei Yu, and Hongrun Wu. A neighborhood comprehensive learning particle swarm optimization for the vehicle routing problem with time windows. Swarm and Evolutionary Computation, 84:101425, 2024.

[11] Korhan Karabulut and M Fatih Tasgetiren. A variable iterated greedy algorithm for the traveling salesman problem with time windows. Information Sciences, 279:383–395, 2014.

[12] Oskari Lähdeaho and Olli-Pekka Hilmola. An exploration of quantitative models and algorithms for vehicle routing optimization and traveling salesman problems. Supply Chain Analytics, 5:100056, 2024.