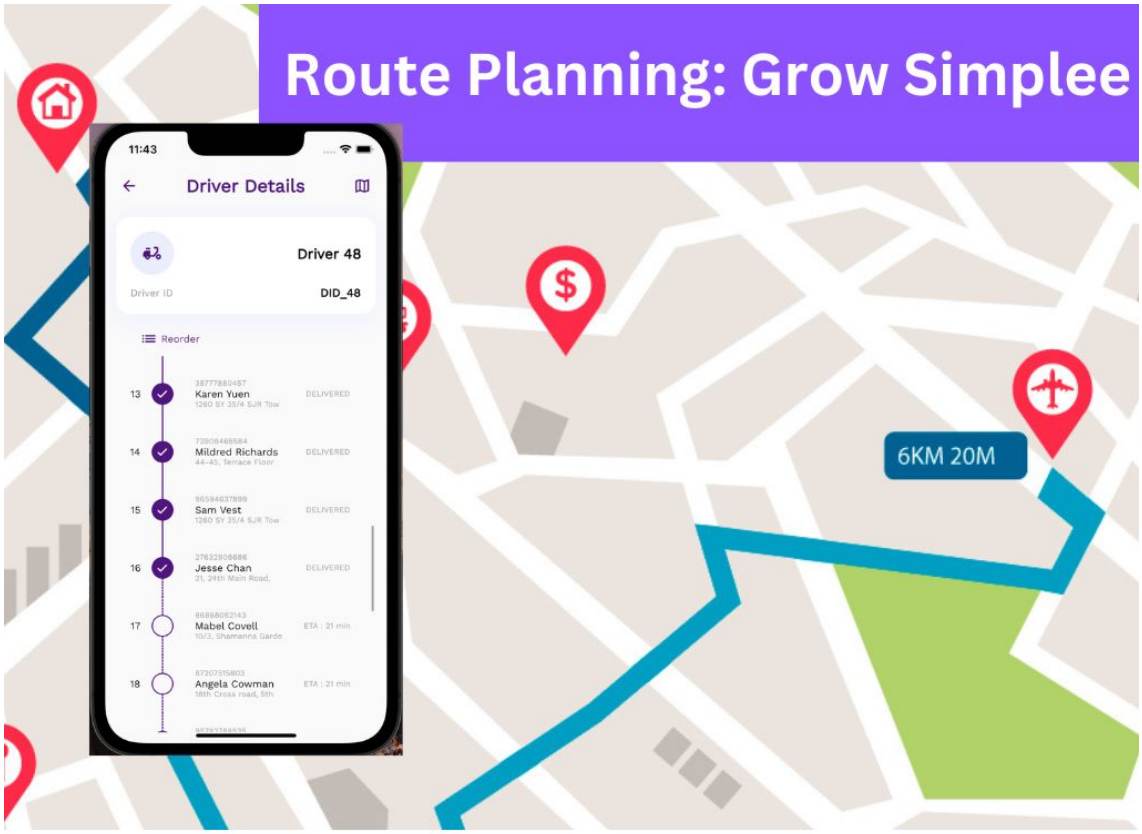
Grow Simplee Final Report

Team No: 41

February 7, 2023

Route Planning Optimisation



Contents

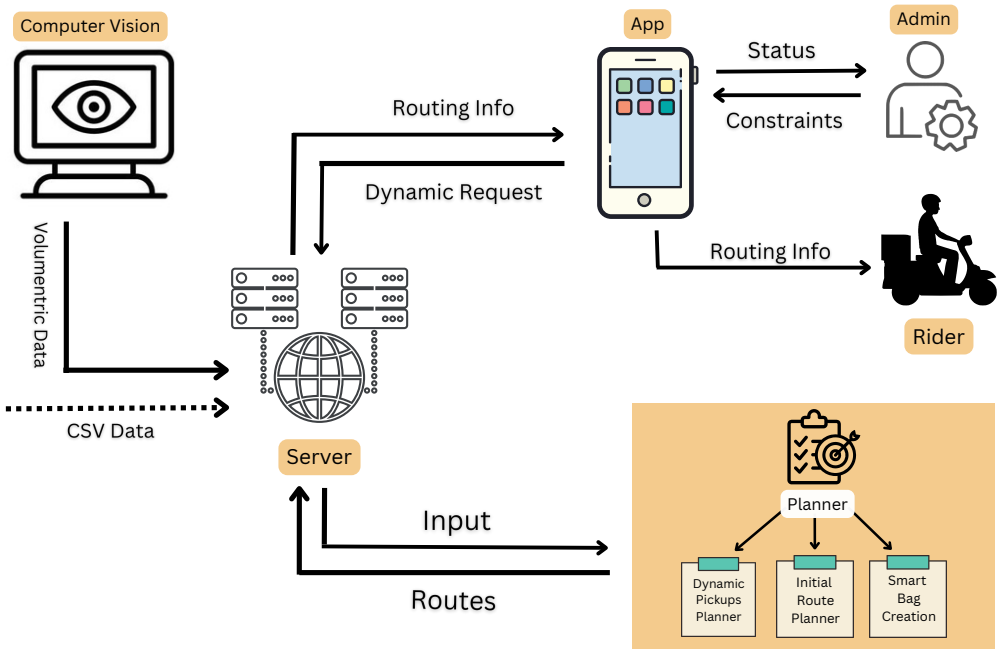


Figure 1: Architecture of *last mile hub delivery system*

1 Introduction

This report presents the *last mile hub delivery system*, which deals with delivering a set of items from the hub to the customers using the riders available who can deliver a subset of items (around 20 items) within the given delivery time (around 4-5 hours). It is a very complex system, and we have used several optimization algorithms at each step. The architecture of this system can be explained using Figure 1. This system sits on the *Server* and interacts with each module as described below.

Computer Vision (Challenge 1) The Computer Vision module is responsible for estimating the volumetric weight of objects in real time and detecting any erroneous/defective items based on their dimensions.

Planner (Challenge 2) The *Planner* is a major module in the *last mile hub delivery system* that performs several optimization tasks: First, it uses the ***Initial Route Planner*** to allocate the packages to the riders and find the optimal route for each rider to deliver the allocated packages. Second, it uses the ***Smart Bag Planner*** to place the allotted items in a rider's bag in the order of deliveries while minimizing the total space in the bag. The third optimization task is done by the ***Dynamic Pickup Planner*** to deal with the dynamic pickup request by the users.

Mobile Application The application module facilitates the capability for both the driver and administrator to access and engage with the data stored in the backend. This includes the ability to review and modify package information, view locations on a map and display the status of packages at various points in time.

1.1 Contributions

Here is the summary of our contributions to this problem statement.

Computer Vision

1. **Volume Estimation using *Pixel-Area calibration***: We propose a method for finding volumetric weight of any object in real time with around **90% accuracy**, by using only a single depth camera, hence it's a **cost effective** solution as compared to other Lidar based solutions.

2. **Estimating Dimensions using a *3D reconstruction model***: We have employed a 3D reconstruction model that takes 2 or 3 monocular images as input and generates a 3D Voxelized model of the object in real-time. Then by using convex hull method, we estimate dimensions from the 3D structure with around 85% accuracy. Due to extensive experimental analysis done on the 3D reconstruction model, it could work better in some cases where other Lidar based methods would fail because of lack of texture or wide baselines.

Distance Matrix Computation

1. We created our own API for distance matrix calculation, saving lots of commercial costs from 3rd party services.

2. Our algorithm involves a precomputation to be stored in disk, based on distance matrix generated from clusters. For this purpose, we exhaust all possible transactions available in Microsoft Bing API.

3. The proposed method is highly scalable in both time and cost.

Planner

1. We formalized *Initial Route Planner* as the capacitated vehicle routing problem (CVRP). We used open-source libraries such as *Google OR Tools*, and implemented other algorithms based on Linear programming, Genetic Algorithms, and Reinforcement Learning solutions. We have conducted extensive experiments to test these methods on synthetic data. We found that *Google OR Tools* is easily scaled and superior to all other solutions that we have experimented with when considering the time taken by the solution.

   We also handled several corner cases like, when the number of vehicles is lesser than the minimum required to deliver to all points, by identifying which delivery points the riders should leave. Additionally, we have formulated our optimization problem such that packages with later EDD are dropped under constraint with higher priority.

2. We formulated *Smart Bag Planner* as *3D bin packing*. Smart bag creation optimizes the filling of a single rider's bag so that items are easy to remove, i.e., packages to be delivered first are on the top, and so that volume constraints are satisfied. We implemented the 3D Bin Packing Problem using two heuristics, namely, 3D Best Fit and 3D First Fit Decreasing algorithms (detailed below). We were able to capture the hierarchy of insertion of packages by assigning each package a weight equal to the index of its drop location. This also helped us create enough space in the bag to handle dynamic pick-ups efficiently.

3. We have proposed two solutions to incorporate **dynamic pickups** into riders' existing routes. The first approach returns an optimal allocation of a single package to a rider, by checking the increase in route length for each rider on allocating the package to that rider (using a **TSP variant**), and selecting the rider with the **smallest increase in**

route length. The second approach works with **multiple dynamic pickup** points, using Google OR-Tools with additional constraints to allocate pickups to riders such that the total increase in route length is as minimal as possible.

Mobile Application

1. We have implemented several features for the admin: to view and edit the packages and the drivers assigned, add new pickup points, and simulate the time to view the same at different timestamps.

2. The app uses the flutter framework due to its cross-platform capabilities and extensive library size. We have created every screen from scratch.

3. We created various modules for handling separate tasks, for example, the interactions with the backend and managing data translation between the frontend and back end.

4. We created a separate module for data and state management in the frontend and models for every entity, e.g. driver, customer, package, and location, each having their attributes and functions.

5. Error handling is also implemented in the application in case of errors like network failure.

1.2 Report Overview

In the rest of the report, we describe each component in this system and the methods we used to build them. Section 2 discusses volumetric weight calculation using computer vision techniques. Section 3 describes the *Planner* that we consider an optimization problem. In Section 4, we demonstrate the working of this system using an *App* that we developed using Flutter. Finally, Section 5 concludes this report.

2 Computer Vision (Challenge 1)

The first part of the last-mile hub delivery system involves estimating the volume of objects and detecting erroneous items based on their dimensions. We performed pixel area calibration using a depth sensor and used those parameters to find the volumetric weight of objects. We have also used a machine learning model that generates a voxelized (a voxel is the 3D equivalent of a pixel) reconstruction of the object in 3D. We used this 3D structure to estimate the dimensions of the object. To find the dead weight of the object we used a weighing machine and the output was read using Optical Character Recognition.

2.1 Pixel-Area calibration

For volume Estimation using a depth sensor, we performed pixel area calibration using an OAK-D depth sensor (See fig 2) and used the calibration parameters to find the volume of objects. We have explained this process in detail in the following section.

Figure 2: OAK-D Depth Camera

2.1.1 Approach

We used a depth sensor to get the depth map of the top view of any object we would have information about the height of each point on the top of the object from the ground. If we can somehow find the area covered by the points on top of the object, we could perform an area integral to find the volume of the object.

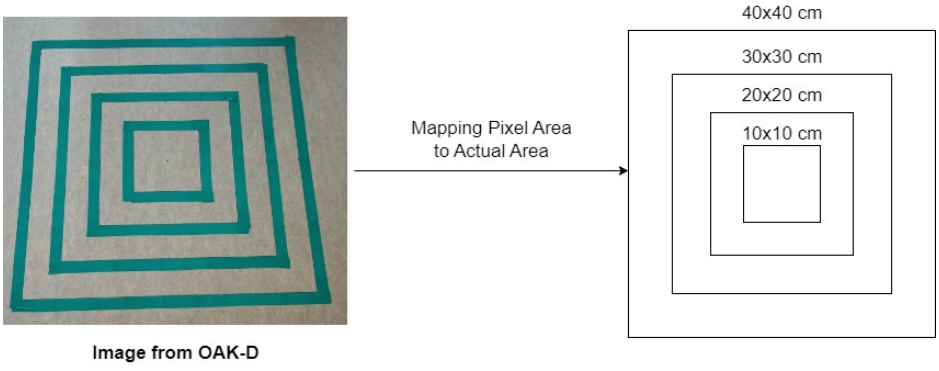


Figure 3: Pixel-Area Calibration

Hence to find the area covered by the points on top of the object we map the pixel area in the RGB image to the actual area covered by the points on top. We do this by marking squares of different dimensions (40x40 cm, 30x30 cm, 20x20 cm and 10x10 cm) on a cardboard sheet (see figure 9) and place the sheet at different heights from the oak-d camera. Then, we

detect contours to find the pixel area in the image and map it to the actual area by fitting a polynomial curve as shown in figure 4.

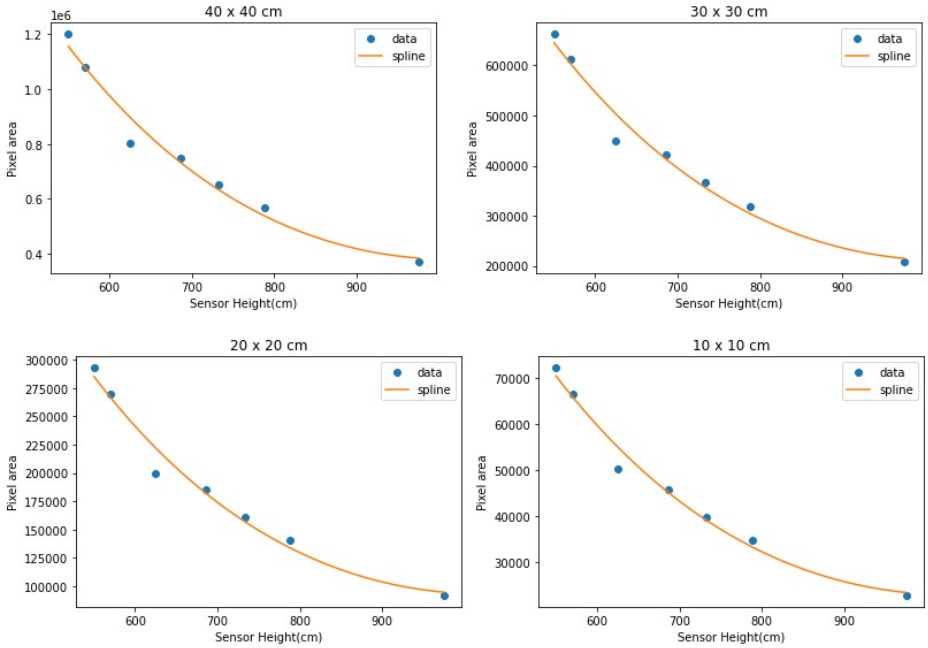


Figure 4: Curve fitting for pixel-area calibration

2.1.2 Results

Object	Actual Volume (litres)	Estimated Volume (litres)	% Accuracy
1	0.9072	0.996	90.17
2	1	1.098	90.2
3	32.275	36.0189	88.4

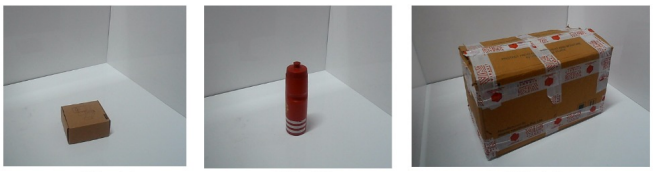


Figure 5: Volume Estimation results

The error in measured volume from the Pixel-Area calibration method was around 10%. One of the reasons is that the least count of depth measured using the OAK-D sensor is around 9mm. With such accuracy, it's difficult to get precise results. Hence if we use a more precise depth sensor, it's possible to get more accurate results.

2.2 3D object Reconstruction

2.2.1 Approach

In this method, volume is estimated from multi-view 2D images as input. We have implemented a paper on 3D Recurrent Reconstruction Neural Network. [1]. In the pipeline, we used pre-trained Neural Network architecture to generate a 3D reconstruction of objects in voxel format from 2D multi-view images captured using 2 monocular cameras. (Figure 6

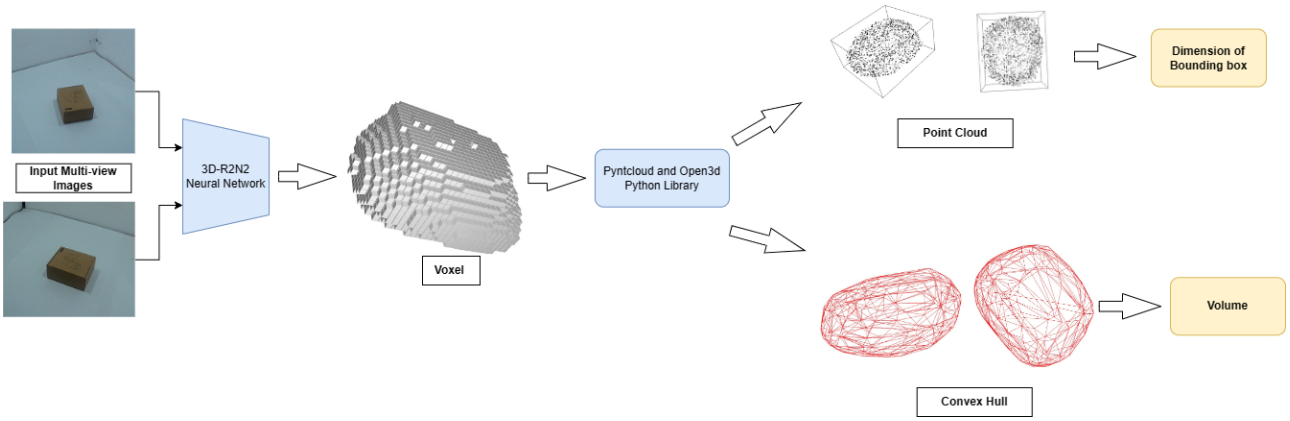


Figure 6: 3D object Reconstruction pipeline

The network takes in 2 images of an object instance from arbitrary viewpoints and outputs a reconstruction of the object in the form of a 3D voxelized object.

We also used pre-existing python library, PyntCloud [2], to generate a convex hull and estimate the volume in the pipeline. We also used Open3D python library [3] to generate the nearest bounding box around the object to get the estimate of the dimensions of the object to recognize erroneous items.

2.2.2 Results

To get an estimate of object dimensions, we calculate a scaling parameter from the known dimensions of an object and the model output. We then use the scaling constant to find the dimensions of other objects as shown in figure 7 and 8.

Known object dimensions (cm)	Model Output	Ratio	Mean
L=19.5	0.506	38.5	36.84
B=9.7	0.2659	36.48	
H=5.3	0.149	35.56	

Figure 7: Calculating scaling constant

Object	Actual length (cm)	Estimated length (cm)	Actual breadth (cm)	Estimated breadth (cm)	Actual height (cm)	Estimated height (cm)	Mean % Accuracy
1	18	20.59	17.6	19.96	17.6	14.86	85.56
2	13.5	15.12	11.2	12.76	6	6.95	86.25


Object 1


Object 2

Figure 8: Dimension estimation results

2.3 Experimental Setup

Our experimental setup consists of 4 white acrylic sheets which provide a suitable background to detect the object kept inside. At the top, we have placed the OAK-D depth camera which captures a depth map from the top. We also place 2 monocular cameras which capture an isometric view of the objects from 2 directions. Figure 9 shows the setup equipped with an OAK-D sensor on top and lights for proper illumination.

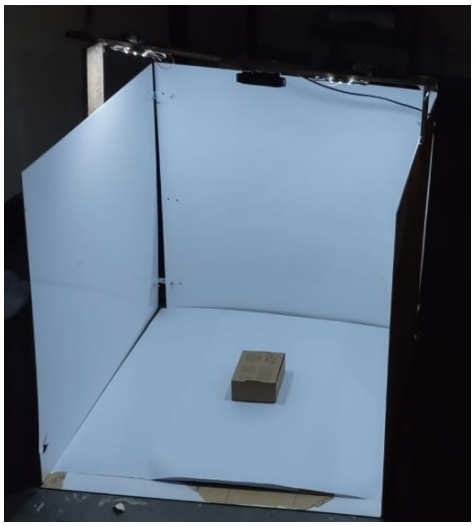


Figure 9: Setup for Volume Estimation

3 The Route Planner (Challenge 2)

The *Planner* is a major module in the *last mile hub delivery system* that performs several optimization tasks: First, it uses the *Initial Route Planner* to allocate the packages to the riders and finds the optimal route for each rider to deliver the allocated packages. Second, it also uses the *Smart Bag Planner* to place the allotted items in a rider's bag in the order of deliveries while minimizing the total space in the bag. The third optimization task is done by the *Dynamic Pickup Planner* to deal with dynamic pickup requests by users. We formalize the first optimization task as capacitated vehicle routing problem (CVRP). The second optimization problem as *3D bin packing*, and the third task as an extension of CVRP The following subsections, detail the CVRP problem, present our solutions, and discuss initial experimental results. In the last subsection, we discuss the second task of dealing with dynamic pickups.

3.1 Distance Matrix

The distance matrix is one of the crucial data required by our routing algorithm. It stores pairwise distance between all the points.

Key Challenge: Computation of distance matrix is very expensive and is $\mathcal{O}(n^2)$. Also, commercial APIs are very expensive if we want to scale our system to a large number of delivery points. In the free version, Google API allows computing only 100 distances in 1 API call. Microsoft Bing API allows computing 625 distances and a total of around 125000 transactions per API key. These are insufficient numbers for computing a distance matrix of say 1000 delivery points, which would require 10^6 entries.

Solution: We have created our own API for computing distance matrix efficiently for Bangalore without additional commercial costs.

The steps are briefly described below:

- We create a road network graph of Bangalore using Open Street Map.
- The graph is now clustered into 1000 clusters.
- A 1000×1000 distance matrix of the clusters is now computed and stored in disk using Bing API, exhausting all the transactions at once.
- When new delivery points are given as input every day, we create their distance matrix using the precomputed cluster distance matrix along with A^* search algorithm on our road network graph to navigate through the clusters.

Major advantage of the solution: Computing distance matrix this way significantly reduces the cost as well as calculates distance with high accuracy.

For example consider the distance between 2 points, $(12.9120799, 77.5745235)$ and $(13.0017902, 77.7231486)$. A trivial solution could be to just use a straight line distance whichs comes out to be $18Km$. The actual distance calculated by Google API is $26.1Km$. Hence, the straight line has an error of around $8Km$, which is very high!

But our solution, approximates it as 26.78 Km, with an error of only 0.18 Km!

Figure 11 and 12 shows the difference in distance calculated between Google API and our solution.

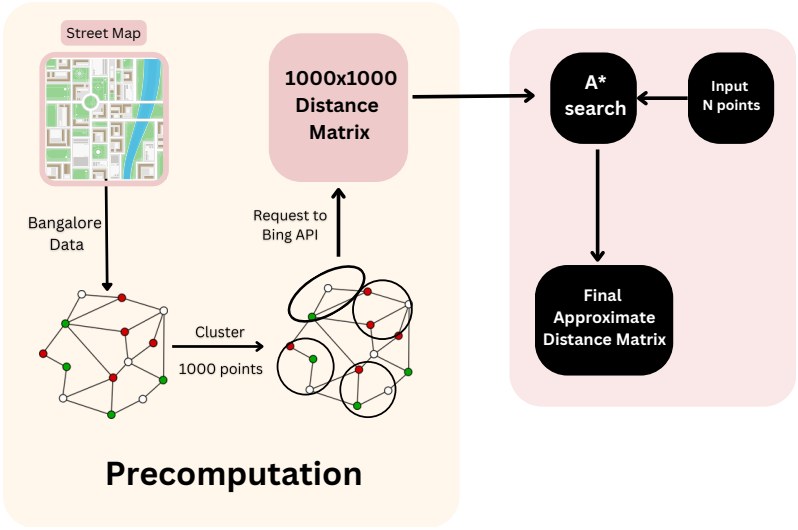


Figure 10: Algorithmic flow for distance matrix computation

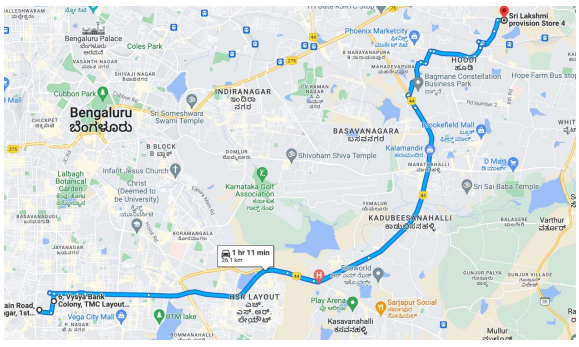


Figure 11: Google API- 26.1 Km

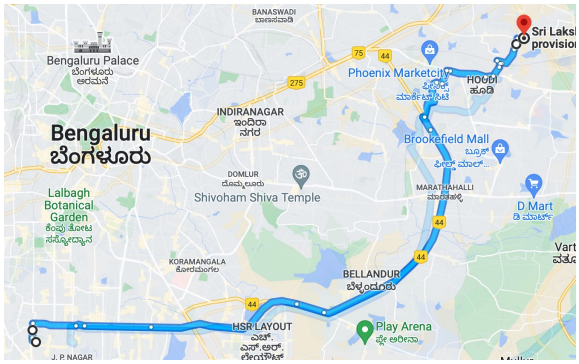


Figure 12: Our Solution- 26.78 Km

3.2 Initial Route Planner: CVRP

CVRP was proposed in [4] as a generalization of the Travelling Salesman Problem (TSP). CVRP is an NP-hard problem that consists of a central hub H, starting from which we need to deliver n items, using m riders, where each rider is constrained to deliver a maximum capacity of C items, with a maximum travel time of T hours for each rider.

CVRP is widely studied, and many approaches exist, from exact, heuristics, and meta-heuristic algorithms to machine learning techniques such as evolutionary algorithms and learning-based algorithms. We considered some of these techniques to perform our initial experiments. *Google OR tools* [5] is an open-source library that has efficient implementations of most heuristic and meta-heuristic algorithms. We also implemented *LP-based solutions* [6] to find the exact solution, and *Genetic Algorithms* [7] which can scale for higher input sizes.

The challenges of each approach, as well as the implementation details, can be found in Appendix B. In the following experimental section, we compare the performance of these approaches on synthetic test cases.

3.3 CVRP Experiments

For testing our route planning algorithms, we created synthetic data with the following assumptions.

- Packet sizes: $3 \times 3 \times 3$ cm^3 to $40 \times 40 \times 20$ cm^3
- Bag capacity: $60 \times 60 \times 100$ cm^3 or $80 \times 80 \times 100$ cm^3
- Average speed of a rider in Bangalore: 20 kmph
- All riders have to complete their delivery in 4hrs, ie. max distance a rider travels is 80 Km, assuming average speed
- Locations are uniform in a square area of 40×40 km^2

Method used	**No. of points**	**Riders used**	**Total (km)**	**Max route length (km)**	**Runtime (sec)**
LP	10	2	**108.766**	62.578	3
	20	3	**179.361**	65.081	1159
	50	-	-	-	-
GA	10	2	112.661	68.457	1
	20	3	189.488	70.671	40
	50	6	343.450	78.254	100
	100	13	725.071	75.451	875
	500	96	6665.475	79.934	4000
OR	10	2	**108.766**	62.578	1
	20	3	**179.361**	65.081	1
	50	4	**291.151**	78.476	1
	100	6	**406.799**	79.867	10
	500	20	**1106.818**	79.526	10
	1000	36	**2117.240**	79.922	10

Table 1: Experiment Results

We used three approaches, namely linear programming (LP), genetic algorithm (GA), and OR-Tools to solve the CVRP over different numbers of locations. Table I shows our results.

N	Algorithms for OR Tools					
	PCA	**PMA**	**SV**	**CS**	**PCI**	**LCI**
10	**108.766**	**108.766**	-	**108.766**	**108.766**	**108.766**
20	**179.361**	**179.361**	-	**179.361**	**179.361**	**179.361**
50	291.151	291.151	-	**277.900**	**277.900**	**277.900**
100	400.470	400.470	401.372	394.187	398.294	**393.027**
500	1041.05	1063.06	1096.32	1114.55	**1040.97**	1099.08
1000	1862.89	**1804.05**	1977.96	1884.90	1828.70	2189.71

Table 2: Comparison of different heuristic algorithms

LP was performed with the *cvxpy* library in Python. While it generated optimal solutions, it was found to take practically unreasonable amounts of time for more than 20 deliveries. The large number of constraints involved causes the time complexity of LP to explode. One possible workaround may be to perform clustering on the dataset and reduce the problem to smaller LP instances.

GA produces close to optimal solutions up to about 20 deliveries but does not perform well on larger datasets. The run-time could be optimized by introducing parallelism via threads. The number of riders shoots up because of capacity constraints, and a better formulation of constraints may bring it down. The total distance could also be improved by adding clustering/locality logic to the objective function.

OR-Tools, having various heuristic solutions at its disposal, clearly outperforms the other two approaches in very less running time. It generates optimal routes for up to 20 deliveries and reaches close to optimality for more points. Fig. 5 displays the routes produced by OR-Tools for a set of 100 uniformly sampled points. We further tested OR-Tools on the test data provided for Bangalore, and we obtained the routes shown in Fig. 6.

As OR-Tools performed the best, we further explored various heuristics available within OR-Tools and compared their performance, as shown in Table II.

We find that **LCI (Local Cheapest Insertion)** outperforms other algorithms for problems of size less than 100, and **PCI (Parallel Cheapest Insertion)** seems to work better on average over problems of all sizes.

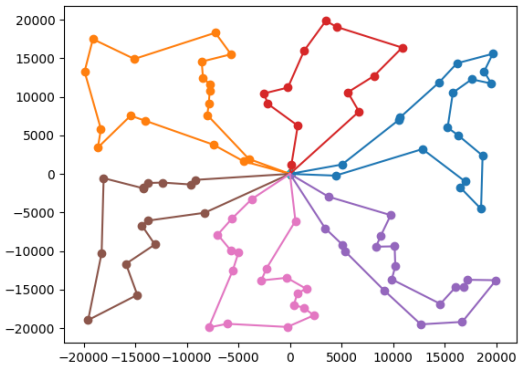


Figure 13: OR-tools route planning output for 100 uniform points

3.4 Dynamic Pickups

We had thought of two approaches, elaborated below.

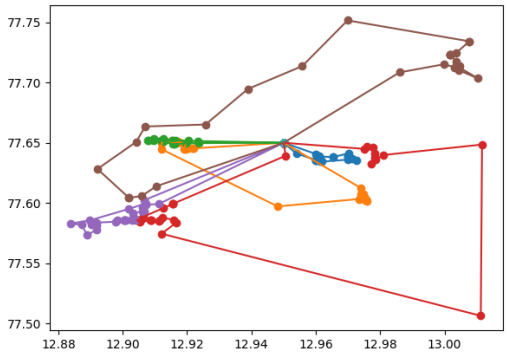


Figure 14: OR-tools route output for actual Bangalore test data for 170 customers, Total Distance travelled: 158.869 Km, Total Vehicles used: 6

3.4.1 Closest rider

When a new pickup location is received, we find the rider currently positioned closest to it, and assign the pickup to that rider. If the remaining bag capacity of this rider is insufficient for the pickup, we assign it to the next nearest rider, and so on. Once the pickup is assigned, we use a TSP solver to recalculate the rider's route.

3.4.2 Least increase in route length

We calculate each rider's new route length if this pickup were to be assigned to him, using a TSP solver, and we select the rider having the least increase in route length. If the bag capacity (at the time of pickup) for this rider is insufficient, we assign it to the rider with the next least increase in route length, and so on.

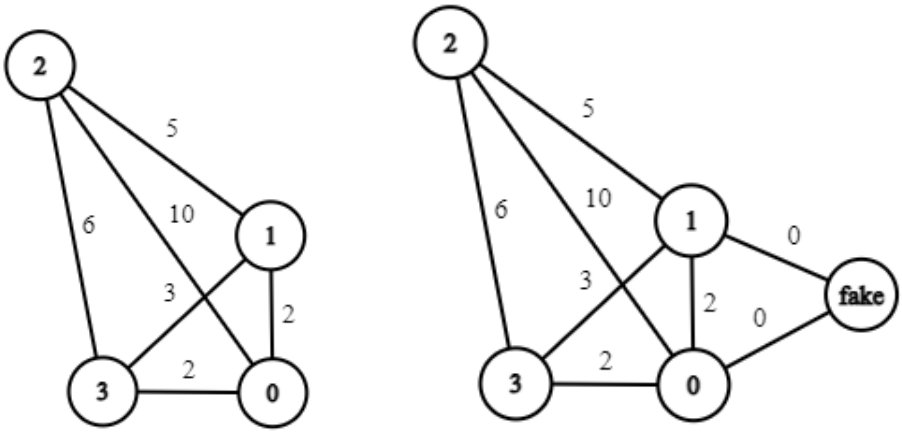
We expect the second approach to generate more optimal routes since it directly relates to the objective while the first approach will produce results faster.

3.4.3 TSP Formulation

We now describe our formulation for the second approach from above.
Since there are many existing efficient TSP solvers, we cast this into a TSP problem.
For each rider, we do the following:

- Let $n-1$ be the number of nodes to which the rider has yet to deliver. WLOG, let node 0 be the hub. Let the n^{th} node be the pickup node.
- We add a fake node numbered $n+1$ which is connected to both the hub and the current node which is the starting point. This node has only 2 edges both of weight 0 connected to the hub and the current node.
- This fake node is used to ensure that the tour is a complete cycle. Because this node is only connected to the hub and the current node, and the rider has to come to and exit from this node, the TSP solution can contain only two possibilities: hub $\rightarrow$ fake $\rightarrow$ current_node or hub $\leftarrow$ fake $\leftarrow$ current_node. If the second case occurs, we can reverse all edges to get a cycle starting at the current node and ending at the hub.
- Since the weights of both edges connected to this fake node are of weight 0, visiting this node does not extend the cycle length but only ensures that the tour is indeed a cycle which satisfies that the first node is the current node and the last node is the hub.

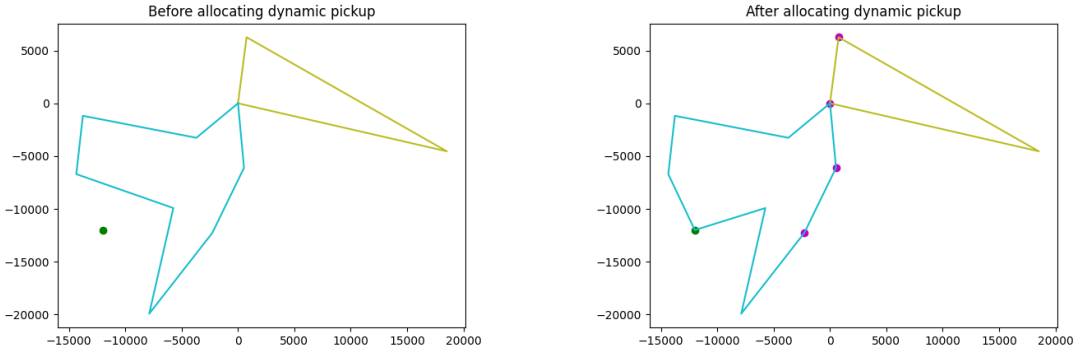
- To ensure that the volume constraints are satisfied, we first travel along the original route until there is sufficient space in the bag for the pickup item. The pickup point is then only connected to those items along the original route where the bag has sufficient space.



(a) Original Graph (b) Casting Into TSP

Figure 15: TSP Formulation

This method is used to handle adding a single new pickup point, as illustrated in Figure 16. Adding multiple pickup points can be handled by iteratively calling this method.



(a) Before allocating pickup (b) After allocating pickup

Figure 16: An illustration of dynamic pickup allocation for a single package. The green node shows the pickup location. Pink nodes are already visited at the time of allocation.

3.4.4 Handling multiple dynamic pickups simultaneously

The TSP formulation above clearly gives an optimal solution for a single dynamic pickup point, but it is not a very good approach for handling multiple pickups. There may be scenarios where pickup points that showed up earlier may need to be reallocated. This is possible if the assigned rider is not currently on his/her way to a pickup location, and some kind of reallocation of pickups gives a more optimal routing solution than the previous iterative approach.

For this, we used Google OR-Tools. We added constraints in OR-Tools to ensure that the delivery items allocated to the original driver remain allocated to that driver (OR-Tools

has constraints to allow pre-allocation of some locations to fixed riders). Similarly, as before, we introduce multiple fake nodes – one for each rider who is not currently positioned at the hub – and each fake node is connected to both the hub and the rider's current position with a distance of 0 from both. The fake node has distance ∞ from all other nodes to ensure it is not accessible from them. Further, we ensure that capacity constraints on riders' bags are not violated by assigning the volumes of packages at the pickup nodes. Finally, we run the ordinary CVRP solution within OR-Tools on the above setup. The pickup points get allocated to riders so as to minimize the total distance travelled.

3.5 Handling different EDDs and scarce resources

We handle EDDs of different delivery packets such that, the earlier EDDs are given higher priority while leaving out delivery points. We introduce an additional **penalty factor** given to nodes based on their EDD. Later EDDs are given lower penalties due to lower priority.

$$\begin{aligned} Objective = &\sum_{i,j=1}^{N} d(x_i, x_j) + \lambda \sum_{i=1}^{N} (EDD_{max}(x_i) - EDD(x_i) + 1) \\ \lambda = &0.5 * SUM(DistanceMatrix) \end{aligned}$$

3.6 Smart Bag Creation

Smart Bag Creation aims to optimize the filling of packages in the rider's bag according to the planned delivery route. The bag should be filled while considering the volume constraints of the packages and the bag. The order in which packages are inserted into the bag should be conducive to easy removal during the delivery route, i.e. packages to be delivered at locations in the first few stops in the route should be at the top of the bag. We will show that this problem is an extension of the 3D Bin Packing Problem - an NP-Hard problem which is well-researched in literature.

3.6.1 3D Bin Packing Problem

The 3D Bin Packing Problem involves a collection of n items, each with a given volume, that must be packed into a container of a given volume larger than the total volume of the items. We assume that each item P_i has rectangular dimensions w_i, h_i, d_i and must be filled into a container B with dimensions W, H, D. Then, the constraints can be formulated as follows.

$$\sum_{i=1}^{n} w_i \cdot h_i \cdot d_i \leq W \cdot H \cdot D$$

In addition to this, we attempt to capture the hierarchy of insertion of packages by giving each package a weight k_i. We have used the implementation described in [8] and modified it to suit our purposes. This solution attempts to maximise the free space available in the bag, in order to accommodate extra packages dynamically picked up.

3.6.2 Methodology

This solution uses two heuristic algorithms for 3D Bin Packing as described below.

3D Best Fit At every iteration, this algorithm selects a pivot point where the back lower left corner of the next package will be placed. If the package does not fit, it is rotated to try out each of the 6 possible configurations. If none of the configurations fit, the package is sent to the end of the queue of packages.

3D First Fit Decreasing In this algorithm, each package is rotated such that its longest side matches the packing direction. The packages are then sorted in decreasing order according to the longest side and inserted into the bin. If an item does not fit, it is rotated so the second longest side matches the packing direction, and so on.

The solution gives the option to layer the packages according to weight, with the heaviest packages placed below. We use this feature to layer the packages according to how soon their drop location is in the route. We do this by assigning each package a weight equal to the index of its drop location.

3.6.3 Experimental Results

We can see below two cuboidal bags, one of dimensions $60 \times 60 \times 100$ and the other of dimensions $80 \times 80 \times 100$. The items have been packed efficiently to minimize volume wastage. The rider will choose the $60 \times 60 \times 100$ bag.
Note: We start by packing items with a higher weight (later drop location) first so that they are below. Thus, the lower weight items (earlier drop locations) are easily reachable.

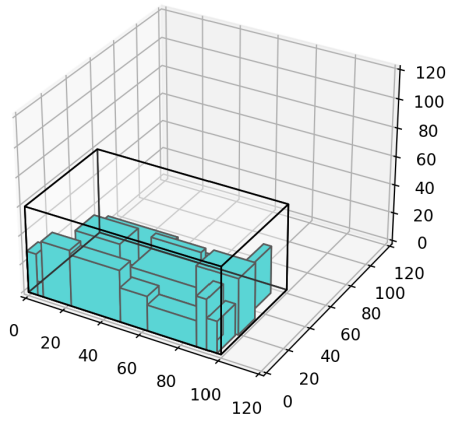


Figure 17: Bag dimensions: $60 \times 60 \times 100$

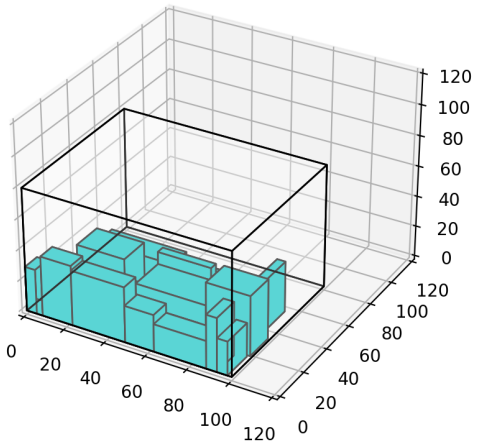


Figure 18: Bag dimensions: $80 \times 80 \times 100$

When we increase the number of items or their size, it can be seen that they fit only in the 80*80*100 bag, not 60*60*100

4 Mobile Application

We have given a mobile application base to our server. Our application consists of two sections - the admin and the rider's sections. The admin section allows the admin to see the status and progress of all the riders, their location and remaining deliveries. The admin can also edit and reorder the packages and their data. We have added the feature for adding any dynamic pickup request or any manual addition of the package from the admin side. In the end, we have a timing simulator where the admin can see an expected simulation of the drivers. In the rider' section, He/She can view the location and route from the included map's screen. The rider's section allows him to see all the packages assigned to him and their details.

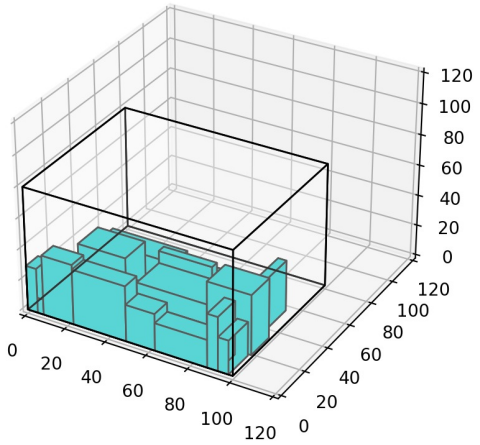


Figure 19: Bag dimensions: $60 \times 60 \times 100$

Figure 20: Bag dimensions: $80 \times 80 \times 100$

4.1 Implementations

We have built a cross-platform application using the Flutter framework. It provides relevant features to the admin and drivers. The backend is built using the Django framework. The front end of the application interacts with the backend using REST API. The backend fetches dispatch and pickup addresses and customer details from a csv file, finds the locations' geocodes, and serializes and stores them in a database. We have used Google Maps Geocode API to get the geocodes of the addresses provided in the csv file. When requested by the application (admin's side), the backend runs the routing algorithm, assigns and stores the data in the database, and returns the output to the frontend. All the requests, e.g. adding 16 pickup points, editing and reordering packages and simulating time, are handled by REST API calls. **Tools Used**

- Flutter - provides cross-platform capabilities
- Google Maps, OpenRoutingService and MapMyIndia
- Django for the backend due to its built-in security features and scalability

4.2 Features

4.2.1 Admin

- **Deliveries List** The list of drivers is shown on the home page of the admin section. A single item displays the Driver Name, Driver ID, total packages assigned to the driver, the remaining capacity of the driver's bag, the percentage of deliveries completed by the driver, the address of the next delivery location and the estimated time to reach it.
- **Driver Details** Admin is navigated to this page when he taps an item from the drivers' list. This screen shows details of the driver, like name and driver id, along with the delivery locations in the form of a timeline. A single timeline element contains the customer details and the estimated time to reach the location. Tapping a single tile shows the package details. It also has the option to view all these locations on a map.
- **Package Details** This screen shows the details of the package, its customer and the driver to which it is assigned. It shows details like Order ID, SKU ID, delivery status, EDD, volume, weight, customer name and address and the option to edit the package details.

- **Time Simulation** This feature allows the admin to fast forward or reset the time to see the details of riders and the packages at a different timestamp. The admin can enter the time to fast-forward in a dialogue box. It updates the delivery status of the packages and riders' locations according to the time.
- **Edit Package Details** This page allows the admin to edit the details of a particular package. For example, he can make corrections in the address or customer name.

4.2.2 Driver

- **Login** The driver needs to enter his driver id in order to log in to the app.
- **Delivery List** The home page shows the list of the deliveries assigned to the driver. It contains two tabs, *Pending* and *Completed*, which show the list of pending and completed deliveries, respectively. A delivery tile shows the customer name and address, the order id, the estimated time to reach (in minutes), the delivery status and the option to view it on a map.
- **Package Details** This page shows the next location of the driver on a map and the route to the location from the driver's last location. It also shows the details of the package and customer and has the option to view the route directions to the location in external apps such as Apple or Google Maps.

4.3 Models used in the Backend

We have implemented the following models in our server to handle the database.

- **Location** - It contains the latitude, longitude as well as the string addresses are given in the database
  - latitude - decimal variable
  - longitude - decimal variable
  - address - string address
  - area - string cluster location
- **Object** - It is a generalised object for the package
  - obj_id - Char variable (example - SKU_1)
  - qr_code - Char variable
  - volume - Integer variable (this data comes from the volume estimation part in cm^3)
- **User** - Stores the data for the customer
  - name - Char variable (Name of the user)
  - location - referenced to Location
- **Package** - This table contains the data for the packages, both for packages that needed to be picked up and the ones expected to be delivered
  - pid - Integer variable, stores the AWB number
  - object - references the object in that package
  - driver_assigned - references to the driver, who has been assigned this package
  - user - references to the user whose package is being delivered

  - volume - integer value, the volume of package in cm^3
  - ETA - integer value, estimated time of delivery
  - EDD - Expected Date of Delivery
  - EPT - Expected pickup time, only for packages to be picked up
  - status - Current status of the package, with various states like Delivery Pending, Delivery done, Pickup pending, etc.

- **Driver** - This table contains data related to the driver
  - driver_id - Char field, contains the unique id of the driver
  - last_location - references to Location object, the location of the last package delivered by the driver
  - next_location - references to Location object, the location of the next package to be delivered
  - packages - list of packages to be delivered
  - bag_status - the total occupied volume of the driver's bag
  - on_duty - Boolean variable, signifies whether the driver has been assigned any package
  - return_time - Integer field, signifies when the driver will return back to the hub

We have connected our database with our frontend using REST APIs in Django. For all the queries in the frontend, we have used the features provided by the REST framework and Django filters to optimise our interaction with the database.

4.4 Simulator

We have made a script in python to simulate our application. In this simulator, the first part is to fill the database with the package list using the csv. For volume estimation, we can use the algorithms mentioned in the CV part. Once the database is filled, we have a timer object in our database, not shown above, only used for simulation purposes. We set the timer to zero. From the frontend, the admin can simulate the timer variable. Whenever the request comes, the server will go through each driver to update its state. The server will also look for all the pickup package request that arrived in that time interval.

For example, the timer starts at 9 AM in the morning with time being 0. Let's call this state S_0. Let say the admin increased the time by 45 min, and the app is in some state S_1, here we are defining state with respect to the position of the drivers and the packages that has been delivered and picked up. Then in this interval of 0 to 45 minutes, the server will run through the packages to find out which packages were expected to be delivered (ETA) in that particular time period. And change the driver's current state to a new location. For the pickup packages, let say there was a request for pickup at the 44th minute, then it would have been handled at the 45th minute of simulation and will be allocated to some driver using the dynamic route planner program. Let say, the pickup is expected to be picked up at the 96th minute, Now, let say in the interval 45 - 75 minutes, there were two new pickups. Now, during the processing in the 75th minute, the server will handle the request for all the three packages as the first pickup package has not yet been picked up.

5 Conclusion and Future Work

We have built a system for the *last mile hub delivery system* that consist of several features. Essentially we worked on three major modules, each of which needed several optimizations.

For the CV part, we have tested the three approaches mentioned above for generating depth maps. We worked on methods for generating point cloud data and estimating volume from it. For the optimization part, we tested four approaches - LP-based, Genetic Algorithms, Google OR tools, and Reinforcement Learning technique. We have tested these algorithms using several synthetic test cases. Google's OR-Tools look promising to scale even for 1000 inputs. Also we have built an App that features several functions.

Appendix A CV Methods

A.1 Volume Estimation by Monocular Depth Estimation

A.1.1 Approach

In this approach, volume is estimated from a single view 2D image as input by generating a depth image of the image. In the pipeline, the point cloud is generated from the depth image from a U-net-style neural network trained end-to-end with pairs of RGB and Depth images. The volume is then calculated using pre-existing python libraries by generating a convex hull.

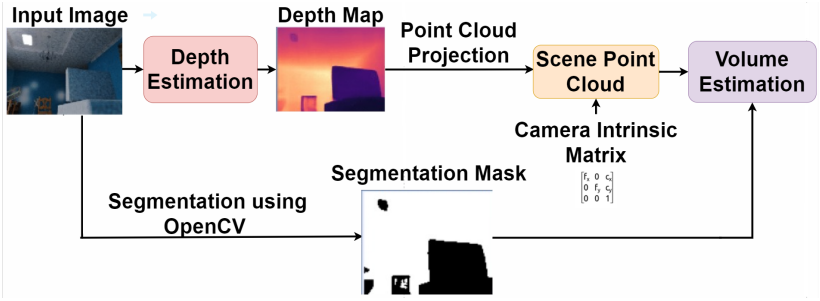


Figure 21: Monocular Depth Estimation

A.1.2 Results

The Depth maps created from this approach were visually good however, the point cloud generated from those depth images was not useful. The estimated volume was always in the order $10^{-}5$. Due to this we had to discontinue that approach.

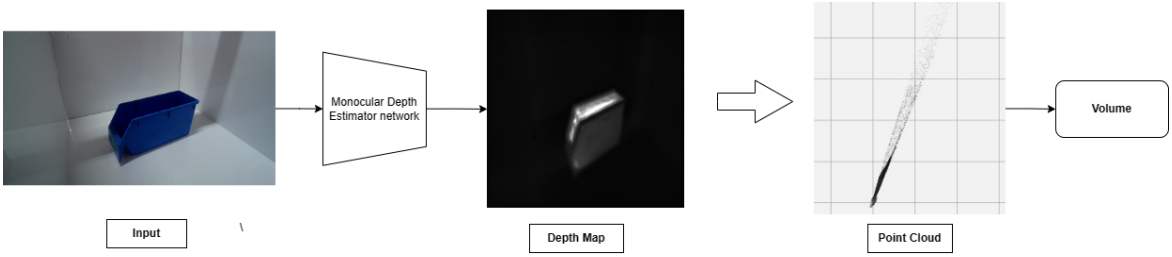


Figure 22: Point cloud generation

Appendix B CVRP Methods

B.1 Linear Programming Formulation

Using a similar LP approach as proposed for TSP in [6], we model this problem with the

following objective and constraints. Consider the following notation:

- $x_{i,j,r}$ – 1 if rider r's tour goes from i to j, else 0
- $c_{i,j}$ – distance between locations i and j
- v_i – volume of package to be delivered at location i
- y_r – capacity of bag carried by rider r

Objective We have locations 0 to n, with 0 being the hub, and riders 1 to m. Given $c_{i,j}$, v_i, y_r, we wish to find $x_{i,j,r}$ that minimizes

$$\sum_{r=1}^{m}\sum_{i=0}^{n}\sum_{j=0}^{n} c_{i,j}x_{i,j,r}$$

Constraints

- There are no self-loops in the graph of locations, i.e., $x_{i,i,r} = 0$ for each $i = 0, 1, 2, \ldots, n$ and $r = 1, 2, \ldots, m$.
- At each delivery location, exactly one rider arrives and departs. For $j = 1, 2, \ldots, n$: $\sum_{r=1}^{m}\sum_{i=0}^{n} x_{i,j,r} = \sum_{r=1}^{m}\sum_{i=0}^{n} x_{j,i,r} = 1$
- The same rider who arrived should depart. For each $c = 1, 2, \ldots, n$ and $r = 1, 2, \ldots, m$. $\sum_{0\le i\le n} x_{i,c,r} = \sum_{0\le j\le n} x_{c,j,r}$
- At most m riders arrive at and depart from the hub. $1 \le \sum_{r=1}^{m}\sum_{i=1}^{n} x_{i,0,r} = \sum_{r=1}^{m}\sum_{j=1}^{n} x_{0,j,r} \le m$
- For each rider $r = 1, 2, \ldots, m$:
  - r can arrive and depart at the hub at most once. $\sum_{i=1}^{n} x_{i,0,r} = \sum_{j=1}^{n} x_{0,j,r} \le 1$
  - The total volume of packages to be delivered by r should be at the most the capacity of r's bag. $\sum_{i=0}^{n}\sum_{j=0}^{n} v_j x_{i,j,r} \le y_r$
  - r's tour, if it exists, must include the hub. $(n+1)\sum_{i=1}^{n} x_{0,i,r} \ge \sum_{i=1}^{n}\sum_{j=1}^{n} x_{i,j,r}$
  - r can travel at most d distance in his tour. $\sum_{i=0}^{n}\sum_{j=0}^{n} c_{i,j}x_{i,j,r} \le d$

With these constraints, the LP solution may still contain isolated 'local' tours. This can be fixed in two ways – subtour elimination or timing constraints. Empirically, we found that subtour elimination performs better. Let S_r denote the set of locations for rider r, where these locations do not form a connected tour. Then, we add the following constraint. $\sum_{r=1}^{m}\sum_{i\in S_r, j\notin S_r} x_{i,j,r} \ge 1$.

Observations Integer linear programming cannot handle a large number of packages, rendering it unfit for practical use.

Pros There are many existing integer programming libraries. We simply plug in the constraints and data to get a solution.

Cons In general, integer programming problems are NP-hard and can take exponential time to solve.

B.2 Genetic Algorithm Approach

This approach, proposed by [7] presents a hybrid **genetic algorithm** combined with a **local search** procedure that can outperform most tabu search heuristics. We can convert a particular instance into an optimal CVRP solution using a splitting procedure. Thus this algorithm is flexible and relatively simple. The reference also mentions that efficient GAs exists for a simpler version (TSP) and an extended version (VRPTW).

The **constraints** that the approach considers are (i) no demand exceeds capacity (to ensure a feasible solution exists) (ii) each rider can return back to the hub (round trip for deliveries) (iii) no rider takes more than 30 deliveries (to ensure load distribution). Along with this, the **objective** that we consider is an appropriately aggregated combination of (i) minimizing the total cost of the trip (travelling) (ii) minimizing maximum trip length (longer routes) (iii) minimizing number of riders required (choosing most efficiently). We found that the optimization and the local search absorb 95% of CPU time. So, we decided to add **parallelism** to the computation using multi-threading. Our main goal is to speed up the local search and design a suitable objective function that gives a better guarantee of the solution being close to optimality.

B.2.1 Problems faced with GA

Genetic algorithms(GA) are not guaranteed to provide an optimal solution or even solutions close to the optimal solution. Sub-optimal CVRP solutions (even slightly off from the optimal solution) result in significantly higher costs and lower efficiency, the effects of which increase with problem size. The inherent stochastic nature of GA requires a large initial population to ensure that the algorithm can model the solution's variations and converge towards the optimal solution. In CVRP, as the problem size increases, the variation in possible routings increases in the order of factorials requiring a considerable population to account for the variations significantly affecting the scaling of the algorithm. Overall the GA was a good enough solution but unable to scale for more large problem instances.

B.3 Google OR tools

Google OR Tools is a collection of open-source software libraries compatible with Python programming language for solving various optimization problems, including the CVRP. OR-Tools has various heuristics and meta-heuristics for routing problems. We focus on two parameters: **first solution strategy** and **local search**.

For the first solution strategy, the following gives promising results: Path Cheapest Arc (**PCA**), Path Most-constrained Arc (**PMA**), Savings (**SV**), Christofides (**CS**), Parallel Cheapest Insertion (**PCI**), Local Cheapest Insertion (**LCI**). And, for the local search, we fix the *guided local search* that was shown to be the most efficient meta-heuristic for the CVRP.

Pros: The main advantage of using Google OR tools over other custom handcrafted approaches is its run time efficiency and ease of use. It can generate approximate solutions for 1000 points in as fast as 10 seconds.

While it may not always provide the optimal solution, it almost always produces approximate solutions that are close enough to optimal for practical use cases.

B.4 Reinforcement Learning Approaches

We have reviewed the literature for three approaches that have attempted to solve the CVRP using RL.

Handling traffic changes using Robust Optimization [9] addresses the CVRP and Traveling Salesman Problem (TSP) using a robust optimization approach, thus accounting for uncertainty present in data, specifically uncertainty about distances between delivery points or nodes.
Pros Robust optimization provides performance guarantees despite frequent environment changes, such as changing traffic conditions. The RL agent is provided with a new problem instance in each episode, and thus learns strategies to generate good solutions for an entire class of instances, and once training is complete even new instances can be solved instantly.
Algorithm The problem is formalised as

$$\max_{y \in Y} \min_{u \in U} f(y, x, u) = \max_{y \in Y} \hat{f}(y, x, U)$$

where known parameters x, an uncertainty set U, and the solution space Y are given as input, and the value of the solution f is considered to be fixed. The "robust objective" $\hat{f}$ determines the minimum value of solution y that can be guaranteed regardless of unknown parameter vector u, and the objective is to maximize this guaranteed value.
RL agents are trained for maximization of the above equation, and each action of the agent is a step to construct the solution. The state space S consists of states s which represent both the given problem instance and the solution constructed so far. The action set A represents solution construction steps such as addition, deletion, and swapping of nodes. Pairwise node distances, representing distances between delivery locations, are uncertain and parameterized by deviation rate α and deviation factor β.

End-to-end framework for VRP [10] develops a framework for solving various VRP problems in which the optimal solution is viewed as a sequence of decisions, thus obtaining near-optimal solutions by increasing the probability of desirable sequences being encoded.
Pros Once the model is trained, it can be used several times without re-training, for any problem generated from the training distribution. The method is computationally superior & scalable.

Timing windows using roll-outs [11] offers an approach to the CVRP. This approach allows the user to tune solution quality at the cost of computation time while being generalisable to other problems. Here, the feasibility of a vehicle to serve a customer in time is given a value using Reinforcement Learning. The vehicle-customer pairs with the top k values are identified. Rollouts are carried out on these pairs using a pre-trained policy. The route with the lowest total distance is identified and optimised in two steps using satisfiability solvers.
Pros Offers an RL solution to CVRP with comparable solution quality and computation times to other heuristic and meta-heuristic-based solutions.

B.4.1 Reinforcement Learning Experiments

Below we show one batch of results obtained by the model described in [10] (as reproduced by us after training it partially on some random instances) for 10 input points.
We give some details about the development and training environment.

Algorithm: We first set up encoder (1D convolution), attention (combination of tanh and softmax) and pointer (GRU combined with dropout) layers. The actor is a deep RL model having both a static and dynamic encoder, which produces tour indices along with the likelihood of it being on the tour(s) (negative log). The critic analyses the complexity based on the log probabilities.

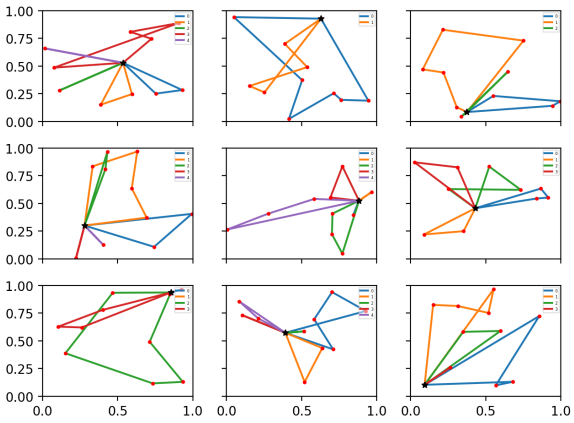


Figure 23: RL experiment results (10 delivery points)

Dataset: The assumptions used when creating the data are (1) Each city has a demand in $[1, 9]$, which must be serviced by the vehicle (2) Each vehicle has a capacity (depends on the problem), and must visit all cities on tour (3) When the vehicle load is 0, it must return to the depot to refill. The static set is the set of locations of the points, and the dynamic set is the set of (load, demand) pairs for drivers and the points respectively.

Hyperparameters: The hyperparameters include (1) maximum load for a vehicle (2) maximum demand (3) Actor and Critic learning rates (4) Batch size (5) Hidden and Dropout size (6) Number of layers and Randomizer seed (7) Train and validation size.

Training: The updates handle valid and invalid states, and the dynamic updates handle whether the driver is able to satisfy maximum demand in a city visit (tour) before returning to the depot. The reward given also considers the Euclidean distance covered by the driver. The training proceeds as (1) actor forward pass (2) reward accumulation (3) critic estimation (4) computing losses (5) backpropagation of actor and critic.

References

[1] Christopher B Choy, Danfei Xu, JunYoung Gwak, Kevin Chen, and Silvio Savarese. 3d-r2n2: A unified approach for single and multi-view 3d object reconstruction. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VIII 14*, pages 628–644. Springer, 2016.

[2] `https://pyntcloud.readthedocs.io/`,.

[3] `http://www.open3d.org/`,.

[4] G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Manage. Sci.*, 6:80–91, 1959.

[5] Laurent Perron and Vincent Furnon. Or-tools.

[6] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.

[7] Christian Prins. A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research*, 31:1985–2002, 2004.

[8] Erick Dube, Leon Kanavathy, Leon K@i, and Owave Za. Optimizing three-dimensional bin packing through simulation. 01 2006.

[9] Tobias Jacobs, Francesco Alesiani, and Gulcin Ermis. Reinforcement learning for route optimization with robustness guarantees. pages 2592–2598, 2021.

[10] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence V. Snyder, and Martin Takáč. Reinforcement learning for solving the vehicle routing problem, neurips, 2018.

[11] Harshad Khadilkar. Solving the capacitated vehicle routing problem with timing windows using rollouts and max-sat, 2022.