Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Shipping optimization for an online store using Google OR-Tools

Technical documentation

V.1.0.0

Semester 5 Project
School of Engineering and Architecture of Fribourg
Department of Computer Science
1 Febuary 2024

Author
Noah Godel

Supervisor
Frédéric Bapst

Document version

N° revision	Date	Description
0.1.0	7 January 2024	Initial version
0.2.0	30 January 2024	Finnished Implementation chapter
1.0.0	1 Febuary 2024	Final version

Contents

1 Introduction

This report presents the activities carried out during the semester 5 project of the Software Engineering Bachelor programme at the School of Engineering and Architecture of Fribourg. The next section provides an overview of the project. For further details, please refer to the specification. The project repository is located at the following URL. Please note that the project is confidential and can only be accessed by authorised users.

https://gitlab.forge.hefr.ch/noah.godel/23-ps5-shipping-optimization-ortools

1.1 Context

As stated in the specification document, Enterprise-XY aims to internally handle certain deliveries of their orders instead of relying solely on third-party deliveries. Currently, they prepare boxes for delivery to warehouses and then transfer them to third-party services for the final delivery.

The objective of this project is to develop an application that can plan the delivery of orders of one day. The first step is to determine the cost-effectiveness of direct delivery, without involving third parties. Once this is established, the application will plan the most efficient routes for the delivery vans to fulfil all orders that can be delivered directly. Evaluation of the precision and effectiveness of the results of the application is also an essential part of the project.

During a previous semester project, a student embarked on the creation of a rudimentary solver that was not optimized. The solver followed a greedy strategy where a van exclusively handled orders for a single ZIP code (although there could be multiple vans assigned to one ZIP code). The vans would then be filled with orders, while ensuring that the time window constraint was met. Afterwards, any vans that incurred a cost higher than the hub delivery cost were eliminated from consideration.

The main focus of the project, however, was to create a script that generates a distance matrix between ZIP codes. Once a functional prototype is developed, it can be compared to the previous implementation.

1.2 Goal

The main objective is to develop a solver that utilizes the data from Enterprise-XY to schedule the trips of the vans. Subsequently, we evaluate the outcomes against the previous implementation and make adjustments to optimize the solution within a reasonable timeframe. Additionally, we devise a strategy to address the actual problem of having orders with real addresses. If time permits, we can also incorporate a visual representation on a map to assist the drivers and initiate the implementation of the real problem.

Initially, we develop an initial prototype that omits certain constraints but utilizes the available data to produce a deliverable. Subsequently, we enhance the prototype and incorporate the remaining constraints to create a fully functional application.

1.3 Documentation Structure

The layout of this document is as follows.

- Introduction: This section provides an overview of the documentation.
- Analysis: This section explains the problem, the data, constraints, and technological decisions, such as the chosen stack.
- Design: This section describes the internal structure of the application and the output data.
- Implementation: This section provides details on how the application was implemented.
- Conclusion: This section discusses the progress of the project, what was implemented, what was not, and the next steps.
- Honour Declaration: This section includes a statement that declares that this document is not plagiarism.

2 Analysis

This section offers a description of the issue and its related data, encompassing the frameworks and other pertinent details.

2.1 Data Overview

As stated above, we possess a compilation of customer-made purchases. Each purchase may consist of multiple boxes. The sole location required for order delivery is the customer's home ZIP code. Regarding orders, the following details are available (in CSV format):

- `Orders_clear_NPA.csv` (and `Orders_clear_NPA_small.csv` that contains a small subset of the other file for testing) have the following columns:
  - *OrderId*: A 256-bit ID for the order (represented as a hexadecimal string).
  - *NumBoxes*: The number of packages (all of the same size) is in order.
  - *CustomerZip*: The ZIP code of where the order must be delivered.
  - *DeliverFrom*: The start of the delivery time window.
  - *DeliverTo*: The end of the delivery time window.
  - *UnloadingSiteId*: The ID (also 256 bits in hexadecimal) of the unloading site (where the order would be delivered if sent via third-parties, see `UnloadingSites.csv`)
- `UnloadingSites.csv` has the following columns:
  - *Id*: The ID of the unloading site.
  - *Price*: The price per box for delivery through this unloading site.
- `OrderWeights.csv` has the following columns:
  - *ORDER_ID*: The ID of the order.
  - *WEIGHT*: The weight in KG of the order.
- `distances_clear_NPA.csv` has the following columns:
  - *NPA4*: The ZIP code.
  - *DISTANCE_BTWN_TWO_DELIVERIES_KM*: Average distance in kilometres between two locations in that ZIP code.
  - *TIME_BTWN_TWO_DELIVERIES_HRS*: Average time in hours between two locations in that ZIP code.

We also possess a matrix of distances between the various ZIP codes. There exists a threshold for distance or time, which determines that only the ZIP codes that can be reached within this threshold are included. The development of the script responsible for generating this distance matrix was a significant component of the previous project. The matrix is stored as a JSON file.

Distance Matrix JSON File structure

```
{
    "1001": {
        "1002": { "duration": 20.0, "length": 12.5 },
        "1003": { "duration": 20.0, "length": 12.5 },
    },
    "1002": {
        "1001": { "duration": 20.0, "length": 12.5 },
        "1003": { "duration": 20.0, "length": 12.5 },
    },
    "1003": {
        "1001": { "duration": 20.0, "length": 21.5 },
        "1002": { "duration": 20.0, "length": 21.5 },
    }
}
```

The duration is in minutes, and the length is in kilometers.
To execute the solver, the values listed alongside the aforementioned files are required.

- The maximum amount of vans that can be used per day
- The maximum amount of boxes that fit in one van
- The hourly cost for using a van
- The ZIP code of the warehouse
- The time needed to unload and deliver the boxes to the client

2.2 Exploration of the Problem

The order data we possess represents the orders placed on a specific day. For each order, we have information regarding the delivery location (ZIP code) and the designated time window for delivery. There are two methods of delivering an order.

The first and more common approach involves shipping the boxes to distribution centres (unloading sites) via trucks. From there, a third party delivers the boxes to customers. This method is known as hub delivery. The second approach, called direct delivery, involves delivering the boxes directly from the Enterprise-XY warehouse using vans. The cost of delivery depends on whether the first or second approach is used. For short distances, direct delivery is likely to be cheaper. The objective of this project is to identify the orders that are more cost-effective to deliver through direct delivery and assign a van to each of these orders.

The vans also need to adhere to the sequence in which the orders should be delivered, taking into account the designated time windows. All the vans used by Enterprise-XY have the same weight and volume capacities, with the weight capacity being the more limiting factor in most cases. Therefore, when assigning orders to vans, these capacities must be taken into consideration.

The route of a van is as follows: leave the warehouse, go to ZIP code 1, go to ZIP code 2, ..., and return to the warehouse. The cost of deliveries depends on the travel time of the van, so minimizing costs entails minimizing travel time. If two consecutive orders in the route are in the same ZIP code, the average distance or time within that ZIP code is added to the cost.

This is how the travel time is calculated for a trip with 2 stops in different ZIP codes (t is the delivery execution time):

$$transit(warehouse, zip_1) + t + transit(zip_1, zip_2) + t + transit(zip_2, warehouse)$$

And here if two consecutive orders are in the same zip code ($\tau(zip)$ is the average time between two locations in the same ZIP):

$$transit(warehouse, zip) + t + \tau(zip) + t + transit(zip, warehouse)$$

The calculation is the same for the distance, except that there is no "delivery execution".

This problem can be expressed as the Vehicle Routing Problem (VRP) for which it is impossible to find the optimal solution in polynomial time. Therefore, we can only hope to find a reasonable solution using Operational Research (OR) libraries that allow us to approximate an optimal solution.

2.3 Technological Framework Overview

Google OR-Tools is an open-source library that allows us to express and solve Operational Research (OR) problems. Provides a set of predefined models for the Vehicle Routing Problem (VRP), which is precisely the type of problem we need to address in this project. The library guides [1] contain various examples that demonstrate how to incorporate features such as dropping visits (optional deliveries that incur a penalty if not made), time windows, and capacity constraints into our VRP formulation. Furthermore, the library is compatible with multiple programming languages, offering flexibility in its usage.

To familiarize oneself with the library, the guide [1], the website that provides insights into the workings of the library [3], and the repository [2] were utilized.

The constraint programming capabilities of Google OR-Tool provide various solvers for different problem types. The routing library utilizes a constraint programming solver to handle variables (representing the values to be found or optimized) and constraints (limitations on the possible values for the variables).

As mentioned above, the Google OR-Tools library provides features to represent the VRP problem. Internally variables and constraints are used to represent the specific problem we are trying to solve. The `RoutingModel` class is responsible for modeling a unique instance of a VRP problem, including its dimensions, variables, and constraints. To keep track of variables and nodes, the library uses indices. However, since the mapping between indices and nodes can be unpredictable, the `RoutingIndexManager` class is introduced. It facilitates the conversion between indices and nodes and needs to be passed to the `RoutingModel` during instantiation. The main variables that are created internally are the following.[3]

- The `next` variables are defined for each node in the model and store the next node in the current route as their value. Thus, if we want to traverse a specific route in the solution, we select a starting node and follow the next variable of this node until we reach the end.
- The `vehicle` variables are also established for every node and include the vehicle index (which can also be interpreted as the route index) to which the node is associated.
- The `active` variable is a boolean variable that is defined for each node and represents whether the node is included in any route of the solution.

The library uses a concept called dimensions to keep track of values that change during the travel of a vehicle. In our case, we use it for distance, time windows, capacities, and the cost of a vehicle. The variables that dimensions create behind the scenes are the following.[3]

- Cumulative variables are variables that are defined for each node and represent the value that has been accumulated up to the current node along the path.
- Transit variables are defined for each node and represent the additional value added to the dimension after visiting the current node. For example, in the case of a distance dimension, it would represent the distance from the location of the previous node to the current node. The value for this variable is determined by the callback specified with the dimension.
- Slack variables are assigned to every node and signify a flexible "gap" along a route. In the case of time as the dimension, the slack dynamic denotes a variable waiting time.

The cumulative variable is computed using the following formula, where i and j represent the nodes and $NextVar(i) == j$.

$$CumulVar(j) = CumulVar(i) + TransitVar(i, j) + SlackVar(i)$$

In order to improve the solver's output, we can optimise the solution by defining an objective function. This can be achieved by implementing a transit callback, such as distance, which evaluates the solution. Subsequently, we can rerun the model with the objective of minimising the objective function. The concept of an objective function also introduces the concept of disjunction, which allows us to make the visit of a node optional. This is done by adding a penalty cost to the objective function for every node that is not visited. If the cost is too high, the penalty will be too expensive and it will always be worth it to visit the node. On the other hand, if the cost is too low, the solver will always choose to drop the node.

To solve a particular model, we can customise the solver by adjusting various configurations, including the following options.[1]

- The time limit for the solver restricts the amount of time allocated for optimizing the solution. This constraint is essential because it is unlikely that the solver will be able to find the optimal solution within a reasonable timeframe.
- The initial solution for optimization is determined using the first solution strategy.
- The local search options is the strategy used for minimising the objective function.

Given the complexity of the data and its well-defined structure, an Object-Oriented Programming language is suitable for this project. Java was chosen as the programming language due to its familiarity. Google OR-Tools has support for Java, and there is even a Maven dependency available for it. Other languages that support the library and were considered were Python and C++.

3 Design

This chapter provides a description of the design decisions made for the application.

3.1 Architecture and Flow of the Application

The application is structured into three major packages, as described below.

- **Model** package includes the data classes that the solver uses.
- **Data** package includes the logic for parsing CSV files and converting them to model objects.
- **Routing** package includes the code to execute the route solver and generate a solution.

Figure 1 depicts the interaction between various packages in the application. Upon launching the application, the parsing functionality of the Data package is utilized to generate the Model objects, which are subsequently employed by the Routing package. The Data package, in turn, reads and parses the specified files.

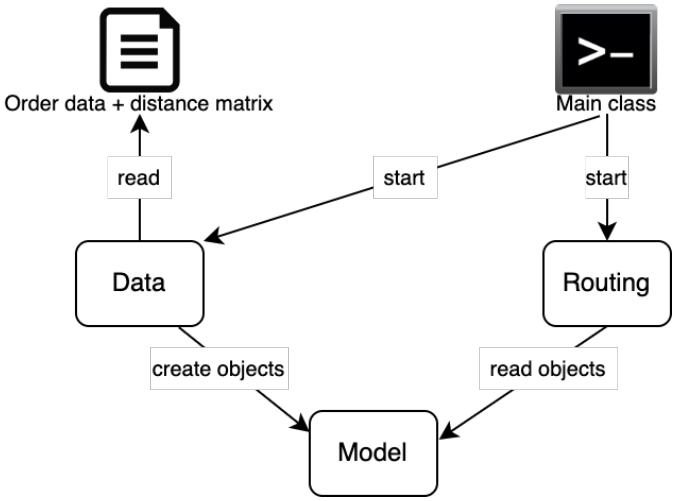

Figure 1: Architectural diagram for application

One benefit of this approach is the loose coupling between the Routing and the Data Packages. Consequently, if there is a desire to introduce an alternative data source, such as a database, in the future, this can be accomplished with ease.

3.2 Error Handling

In general, parsing fails when a file is in an incorrect format, such as having an incorrect number of columns or invalid value types. Specifically, when parsing orders, we also consider it a failure if there is no corresponding weight in the `OrderWeights.csv` file or an unloading site in the `UnloadingSites.csv` file.

The method for dealing with inconsistencies within the distance matrix file is as follows.

- Some ZIP codes used in the distance matrix file are invalid. In this case, we just ignore them and do not insert them into the distance matrix object. When there is no route between two ZIP codes in the distance matrix, the distance value is set to a very high number to indicate that it is effectively unreachable.
- It is normal to have some ZIP codes in the distance matrix file that are not included in the `distances_clear_NPA.csv` file. This is because the distance matrix file is pre-generated and does not have information about all the necessary ZIP codes.

In the Routing package, there are limited possibilities for errors to arise. The only instances where an error may occur are when the Google OR-Tools solver is unable to find a feasible solution within the specified time limit or when no solution is possible, resulting in a null solution. In these situations, we throw and exception to communicate that the router was unable to find a solution.

3.3 Testing Strategy

Testing mainly revolved around the routing package rather than the parsing aspect. To ensure a complete evaluation, two types of tests were conducted. The initial test involved a data set that was entirely controlled by us and had a predetermined solution. The solver was examined to verify whether it returned the expected solution. The second test used the small-order data set, where only a few sanity checks were performed on the solution.

4 Implementation

This chapter provides a description of the project's implementation.

4.1 Order Data Parsing

The first step was the parsing of the data provided by Enterprise-XY. As described in Section 2.1 there are 2 different types of files that have to be parsed: JSON and CSV. The data gets parsed into the data model that will later be used by the solver.

4.1.1 The Data Model

The solver uses the data model as an interface to access the data for the orders that need to be planned. In this project, the data is supplied in JSON and CSV file formats. Therefore, implementing the data model requires parsing these files and generating a distance matrix, which enables querying the distance between any two points.

The data model also contains additional data for routing.

- The hourly cost to operate a van.
- The ZIP code of the depot.
- The maximum number of boxes per van.
- The weight limit of a van.
- The maximum number of vans that can be used.
- The time taken for the delivery of the boxes from the van to a customer.
- The time limit for solving.

To access the orders, the routing solver requires a specific format. The node at index 0 represents the depot, which serves as the starting point for all vans. Afterward, each order is represented by a separate node. In this context, a node corresponds to a stop for the van.

The data model also provides access to the distance matrix. By providing two nodes as input, we can retrieve the distance or time between the locations of these two nodes. In cases where there is no entry in the matrix for the connection between the two locations, a significantly large value is returned. This discourages the solver from utilizing this connection, as it would be more costly than the hub delivery option.

4.1.2 Parsing the CSV Files

There are 4 different CSV files that need to be parsed, containing the following data: the locations (mean distance / time within a ZIP code), the orders, the order weight, and the unloading site price. The process is very similar for all the files (see the listing below).

Parsing of a CSV file with 2 columns

```java
private static Map<String, Float> orderIdFloatMap(String path, String separator)
    throws IOException {
    List<String> lines = Files.readAllLines(Path.of(path));
    lines.remove(0);

    Map<String, Float> zipFloatMap = new HashMap<>();
    for (String line : lines) {
        String[] data = line.split(separator);
        zipFloatMap.put(data[0], Float.parseFloat(data[1]));
    }
    return zipFloatMap;
}
```

The data regarding orders is distributed across several files. To handle this, we initially extract the information from the weights and unloading site files and store it in a map, where the key represents the ID and the value represents the corresponding data. This map is then referenced when parsing an order. Additionally, we want to ensure that there is an entry in the distance matrix for every order. If there is no entry in the weights or unloading site file for a specific order, an exception is also thrown.

4.1.3 Parsing the Distance Matrix

Regarding the distance matrix, the JSON file given contains more entries than the number of locations we have. This is because the file is pregenerated and can be utilized for multiple datasets. In our distance matrix, we only require the ZIP codes that are present in our location data. For the diagonal of the matrix (where the "from" and "to" values are the same), we use the values from the locations file, which represent the average distance or time between two points within a ZIP code.

To perform the parsing, we utilized the "org.json" library to parse the file, which is stored in a single line format.

Parsing of a CSV file with 2 columns

```java
private static DistanceMatrix parseDistanceMatrix(String path, Map<Zip, Pair<Distance, Duration>> locations)
        throws IOException {
    List<String> lines = Files.readAllLines(Path.of(path));
    if (lines.size() != 1)
        throw new IOException("Distance matrix file should contain only one line");

    JSONObject data = new JSONObject(lines.get(0));
    DistanceMatrix distanceMatrix = new DistanceMatrix();
    for (String from : data.keySet()) {
        JSONObject fromData = data.getJSONObject(from);

        // ignore invalid zip codes
        if (Zip.isInvalid(from)) continue;
        Zip fromZip = Zip.of(from);

        // set distance to self to the inter zip distance and duration
        if (!locations.containsKey(fromZip)) continue;
        Pair<Distance, Duration> location = locations.get(fromZip);
        distanceMatrix.setZipData(fromZip, fromZip, location.first(), location.second());

        for (String to : fromData.keySet()) {
            JSONObject toData = fromData.getJSONObject(to);

            if (Zip.isInvalid(to)) continue;
            Zip toZip = Zip.of(to);

            // ignore not needed zip codes
            if (!locations.containsKey(fromZip)) continue;

            // duration is a float representing minutes it needs to be converted to seconds
            // to preserve precision
            long seconds = (long) (toData.getDouble("duration") * 60);
            Distance distance = Distance.ofKiloMeters(toData.getDouble("length"));
            distanceMatrix.setZipData(fromZip, toZip, distance, Duration.ofSeconds(seconds));
        }
    }
    return distanceMatrix;
}
```

4.2 Routing Solver

With the data now parsed, we can now implement the routing solver. The implementation of the constraints is explained first, followed by the configuration of the solver and finally the parsing of the solution.

4.2.1 Distance Dimension

The distance dimension is utilized to monitor the distance covered by a specific van throughout its route. This dimension solely limits the maximum distance that a van can travel during its route. Initially, we establish a callback function with the library, which takes two internal indices, converts them into nodes, and subsequently provides the distance between the nodes in meters as an integer. We then use the pointer to the callback to add a dimension on the routing model.

Distance dimension

```java
final int distanceCallbackIndex = routing.registerTransitCallback((long
    fromIndex, long toIndex) -> {
    Distance distance = data.distance(
            data.routeNodeAt(manager.indexToNode(fromIndex)),
            data.routeNodeAt(manager.indexToNode(toIndex)));
    return Math.round(distance.meters());
});
routing.addDimension(distanceCallbackIndex, 0,  // no slack
        1_000_000,                              // max distance per vehicle is
        not very important for now
        true,                                   // start cumul to zero
        "Distance");
```

Internally, the dimesion will use the callback to calculate the distance between all the nodes along the route of the current van and add them together.

4.2.2 Monetary Cost Dimension

The monetary cost dimension is the main dimension of this model since it is the final value of this dimension that we want to minimise for all vehicles. The calculation of the cost of one van throughout its route is solely based on the travel time for the moment, but this could be changed very easily by changing the callback for this dimension. The implementation of this dimension is very similar to the distance dimension.

In this scenario, we also define the arc cost evaluator for all vehicles. This indicates that this is the primary callback utilized to determine the objective function value. The objective is to minimize the overall financial cost of a solution. Furthermore, we impose a constraint on the maximum cost of a single route (the total value of the dimension on a particular route) to be 8 hours of salary (ensuring no route exceeds 8 hours).

The callback function uses the callback function designed for the time dimension (see Section 4.2.5). It converts time into minutes and then calculates the corresponding monetary amount based on the hourly cost. If we wish to modify the method for calculating the monetary cost, we can make the necessary changes in this callback. For instance, if we want to include the cost of fuel, which is proportional to the distance, we would utilise the distance callback.

Distance dimension

```
// Use the monetary cost as arc cost evaluator for all vehicles.
final int monetaryCallbackIndex = routing.registerTransitCallback((long
↪  fromIndex, long toIndex) -> {
    long durationInMinutes = timeCallback.applyAsLong(fromIndex, toIndex);
    return Math.round(data.config().hourlyVehicleCost() * durationInMinutes / 60);
});
routing.setArcCostEvaluatorOfAllVehicles(monetaryCallbackIndex);

// Add monetary cost dimension.
routing.addDimension(monetaryCallbackIndex, 0,              // no slack
        Math.round(8 * data.config().hourlyVehicleCost()),  // represents the
        ↪  travel cost for 8 hours
        true,                                               // start cumul to zero
        "MonetaryCost");
```

4.2.3 Capacity Constraint

The next constraint involves the fact that vans have limited capacity for the number of boxes and the total weight of the boxes. The implementation of this constraint is similar to the previous ones with the difference that the callback is unary. We don't need to know what the previous node in the route was, we only need to know the amount of boxes and the weight of the order of a node.

The order in which the dimensions are represented seems to be reversed. Instead of decreasing along the route, the cumulative value of the dimensions actually increases as the route progresses and reaches the maximum capacity of a van. Due to the limitations of the library, we cannot set a specific starting value for the cumulative value other than zero. Therefore, we have inverted the logic by representing the variable as the remaining capacity. This variable increases with each delivery made and cannot exceed the maximum capacity of the van.

Distance dimension

```
// Add boxes dimension.
final int boxesCallbackIndex = routing.registerUnaryTransitCallback((long index)
    -> {
    RouteNode node = data.routeNodeAt(manager.indexToNode(index));
    if (node.isDepot()) return 0;
    return node.order().numberBoxes();
});
routing.addDimension(boxesCallbackIndex, 0, // no slack
        data.config().maxBoxesPerVehicle(), // maximum boxes for all vehicles
        true,                               // start cumul to zero
        "Boxes");

// Add weight dimension.
final int weightCallbackIndex = routing.registerUnaryTransitCallback((long index)
    -> {
    RouteNode node = data.routeNodeAt(manager.indexToNode(index));
    if (node.isDepot()) return 0;
    return Math.round(node.order().weight().grams());
});
routing.addDimension(weightCallbackIndex, 0,                     // no slack
        Math.round(data.config().maxWeightPerVehicle().grams()), // maximum
        weight for all vehicles
        true,                                                    // start cumul
        to zero
        "Weight");
```

4.2.4 Dropping Visits

We also aim to prioritize direct delivery for orders that are closer to the warehouse. This implies that we are considering the nodes as optional, but if they are not visited, we will incur the cost of hub delivery. Given that our objective function is the total cost of all routes, we can incorporate a disjunction where the penalty for not visiting a node is equal to the hub delivery cost. As a result, the objective function value accurately reflects the overall cost of the solution.

Distance dimension

```
// add hub delivery cost disjunction
for (int i = 1; i < data.numberRouteNodes(); ++i) {
    long hubDeliveryCost =
    Math.round(data.routeNodeAt(i).order().totalHubDeliveryCost());
    routing.addDisjunction(new long[]{manager.nodeToIndex(i)}, hubDeliveryCost);
}
```

4.2.5 Time Windows Constraint

To enforce the time window constraint, we initially establish a time dimension that measures time in minutes per day. The initial value of the cumulative variable is not fixed at zero, but rather varies. The departure time of a van corresponds to the cumulative variable at the starting node of a given route. The subsequent values are computed using a travel time callback (for a detailed explanation of the travel calculation, please refer to Section 2.2).

Next, we impose further limitations on the variables related to the time dimension. Specifically, we limit the cumulative variables' values at each node to fall within the time window specified for the order. Given that our time dimension represents the time of day, we can easily convert the time window boundaries to minutes. As a result, we set the range for the cumulative variable to be between the start and end times of the time window.

Time dimension

```
// Add Time dimension.
LongBinaryOperator timeCallback = (long fromIndex, long toIndex) -> {
    RouteNode from = data.routeNodeAt(manager.indexToNode(fromIndex));
    RouteNode to = data.routeNodeAt(manager.indexToNode(toIndex));

    Duration duration = data.duration(from, to);

    if (!to.isDepot())
        duration = duration.plus(data.config().deliveryExecutionTime());

    return Math.round(duration.toSeconds() / 60.0);
};
final int timeCallbackIndex = routing.registerTransitCallback(timeCallback);
routing.addDimension(timeCallbackIndex,
        0,          // no slack
        24 * 60,    // no max time per vehicle (set to 24 hours)
        false,      // start cumul to zero
        "Time");

RoutingDimension timeDimension = routing.getMutableDimension("Time");
// Add time window constraints for each location except depot.
for (int i = 1; i < data.numberRouteNodes(); ++i) {
    long index = manager.nodeToIndex(i);
    Order order = data.routeNodeAt(i).order();
    timeDimension.cumulVar(index).setRange(order.timeWindow().startAsMinutes(),
            order.timeWindow().endAsMinutes());
}

return timeCallback;
```

4.2.6 Solution parsing

After the solver provides a solution, which is in the form of an assignment (if we consider a complete graph where the vertices represent the nodes, then a solution, or assignment, would be another graph with the same vertices but not all the arcs), we convert it into a more suitable format for further processing. An assignment of a model has values that we can consult for all the variables.

To begin with, we go through each route and gather the nodes of the route in a sequential list. During this process, we gather various details such as the overall distance of the route, the starting and ending times, the route's cost, and a list of departure times at each stop. We subsequently compile all these routes into a list, gather all dropped orders, and pass the lists to the constructor of the `RoutingSolution` class.

4.3 Testing and CI

To test the routing solver, two types of tests were employed. These tests are executed in a Gitlab Continuous Integration (CI) pipeline for every commit to ensure that there is no erroneous code in the repository. In the following section these 2 types of tests are described.

4.3.1 Simple Model Test

To perform the initial test, we run the routing solver on a straightforward and controlled data set. The solution given by the solver is predictable, and we confirm that this is indeed the case.

The dataset contains 7 orders, each with a unique ZIP code. The last two orders have a significant weight and a large number of boxes, respectively. The distance matrix was created in such a way that there are only two possible routes for the two vans to deliver four out of the seven orders. Although the path to the last ZIP code is included in the matrix, it is located far away and therefore not considered worthwhile to visit. The 2 orders that have large wieght and a big number of boxes are beyond the capacity of the vans so they get dropped.

The evaluation of the anticipated solution and the actual outcome is conducted as follows. Initially, we examine the routes and verify that there are indeed two vans with two orders each, in the expected sequence. Subsequently, we verify the accuracy of the total cost in the solution. Alongside these checks, we also perform the same superficial tests that are conducted in the alternative type of test (refer to the subsequent section for further elaboration).

4.3.2 Small Dataset Test

In this test, we utilize the order data from the `Orders_clear_NPA_small.csv` file, which is a subset of the data provided by Enterprise-XY. The solver is applied to this data, and some preliminary tests are conducted on the obtained result. Due to the unpredictable nature of the result, a comprehensive test cannot be performed.

However, the following tests are conducted on the result:

1. Verification of the presence of all orders in the solution.
2. Confirmation of an overall positive profit.
3. Assurance of positive profit for each individual route.
4. Adherence to volume and weight constraints.
5. Compliance with time windows.

4.3.3 Automated tests with CI

In order to streamline the testing process, a GitLab CI configuration was set up. This configuration enables the automatic building and testing of the project using Maven. However, these tasks are only initiated if there have been modifications made to the source code of the routing solver application.

gitlab-ci.yml

```yaml
default:
  image: maven:3.9-eclipse-temurin-17

stages:
  - build
  - test

mvn-build:
  stage: build
  script:
    - cd routing_app
    - mvn compile
  only:
    changes:
      - routing_app/**/*

mvn-test:
  stage: test
  script:
    - cd routing_app
    - mvn test
  only:
    changes:
      - routing_app/**/*
```

4.4 Results

In order to assess the performance of our application in relation to the previous version, a series of tests were conducted. These tests were conducted using both the small and full datasets, and with two different configurations. The first configuration was set using Mr. Bapst's recommended default values, while the second configuration is the default used in the previous project's context.

Configuration 1

```
private static final int DEFAULT_VEHICLE_NUMBER = 30;
private static final float DEFAULT_HOURLY_VEHICLE_COST = 200;
private static final int DEFAULT_MAX_BOXES_PER_VEHICLE = 64;
private static final Weight DEFAULT_MAX_WEIGHT_PER_VEHICLE =
↪  Weight.ofKiloGrams(1000);
```

Configuration 2

```
private static final int DEFAULT_VEHICLE_NUMBER = 200;
private static final float DEFAULT_HOURLY_VEHICLE_COST = 30;
private static final int DEFAULT_MAX_BOXES_PER_VEHICLE = 64;
private static final Weight DEFAULT_MAX_WEIGHT_PER_VEHICLE =
↪  Weight.ofKiloGrams(200);
```

The measurements that were made on the tests are the duration of the execution of the programm and the profit of the solution.

Dataset	Configuration	Profit new	Duration new	Profit old	Duration old
small	1	389.-	30s	20.-	6s
full	1	675.-	5min	97.-	9s
small	2	2'800.-	30s	1'500.-	6s
full	2	15'400.-	5min	12'700.-	8s

Table 1: Test results

In our tests, it is evident that the new application consistently outperforms the old application. It is worth mentioning that the difference in profits between the two configurations is smaller for the second configuration compared to the first one. This suggests that we may be approaching an optimal solution, and it is possible that the old application also came close to it. However, for the other configuration (where fewer orders are delivered directly), the new implementation significantly outperforms the old one in finding a more optimal solution.

It is important to consider the execution times as well. The execution time is adjustable in the new implementation, and the shorter it is configured, the less optimized the solution becomes.

5 Conclusion

The application is functioning properly, and with the order data provided in CSV files, we can create routes for direct delivery for a portion of the orders. However, as described in Section 5.1.3, there is a limitation in the time dimension that prevents us from incorporating flexible breaks of varying lengths during the delivery process.
As stated in Section 4.4, our application demonstrates superior performance compared to the previous version across all conducted tests. However, it should be noted that the new application operates at a slower speed and requires more resources to run.
Throughout this project, we conducted an analysis of the problem, parsed the data provided by Enterprise-XY, established our constraints and variables, and assessed the outcomes of the application.

5.1 Challenges

Throughout the duration of this project, we encountered various challenges, the majority of which were successfully resolved. In this section, we will provide an explanation of these challenges and their resolution.

5.1.1 Disjunction and solver settings

During the implementation of the disjunctions, a problem arose where regardless of how they were implemented, they were being ignored and had no impact on the solver's result. Disjunctions are not considered "hard" constraints that determine the validity of a solution, but rather a means to achieve a more optimal solution (by reducing the cost function). As the work had so far only focused on "hard" constraints, the solver was configured to simply search for the first solution that satisfied all constraints, without performing any optimization. Consequently, when we started implementing the dropping of orders, they were disregarded because the solver was solely focused on finding a valid solution, rather than optimizing it.

We implemented a straightforward solution by incorporating a local search metaheuristic configuration into the solver. This approach involves utilizing a technique to explore alternative solutions that are similar to a given valid solution, with the aim of finding more optimal solutions.

5.1.2 Depot deliveries

During the experimentation process, Mr. Bapst, the project supervisor, identified a bug related to orders within the same ZIP code as the warehouse. The bug caused incorrect calculation of the distance and duration for routes delivering these orders.

The reason behind this bug is as follows: When an empty route exists, the solver represents it as a route from the warehouse to the warehouse. In such cases, it is necessary to have a distance and duration of 0. Previously, the distance matrix was responsible for handling this, but it only considers ZIP codes and not the route nodes, which can be either an order or the warehouse's ZIP code. Consequently, it cannot distinguish between an order in the same ZIP code as the warehouse and the warehouse itself.

To address this issue, the solution involved transferring the logic of returning a 0 for transit from the warehouse to the warehouse to the data model, which has access to the route node objects.

5.1.3 Slack variables in the time dimension

During the project, it was observed that the implementation of the time dimension did not incorporate the use of slack variables. Slack variables are intended to enable flexible "breaks" of limited duration between two nodes. These breaks are crucial for accommodating time windows, as it may be necessary to wait at a specific location before making a delivery to ensure it falls within the designated time window.

Although this issue was identified late in the project, the application remains functional. Integrating slack variables into the implementation could potentially lead to more optimal solutions by improving the handling of time windows. However, there are challenges in adapting the code to incorporate slack variables. The cost dimension relies on the transit time callback to calculate values. Introducing slack variables would mean that the time taken for a route is no longer solely determined by the transit callback, and could be longer. Consequently, the total cost of a specific solution can only be determined once values for the slack variables have been obtained.

An attempt was made to replace the monetary cost dimension with a finalizer variable, which would be minimized only after a valid solution was found. However, this approach resulted in the solver returning solutions with negative profits. The exact reason for this failure is unknown, but it is hypothesized that the introduction of slack variables introduces too many unknowns, making it difficult for the solver to reliably find good solutions within a reasonable timeframe.

5.2 Future Work

In addition to incorporating slack variables into the implementation, there are several other features and tasks that could be beneficial as future steps for the project.

5.2.1 Modification of the monetary cost function

At present, the monetary cost is denoted by a dimension. Its values are determined using the transit callback of the time dimension. However, this method of calculating the cost of a route is limited and often does not accurately reflect reality. In reality, we typically have time-dependent costs such as driver salaries, as well as distance-dependent costs like maintenance and fuel expenses.

To address this, we can modify the current implementation by introducing an additional parameter in the configuration that represents a cost per distance. This parameter can then be incorporated into the calculation using the distance transit callback. However, it is important to note that the implementation of the monetary cost may be impacted by the presence of slack variables (refer to section 5.1.3).

5.2.2 Visualisation of the solution

An additional valuable functionality could involve displaying a graphical representation of a solution on a map. Given that a JSON serializer has already been incorporated for the solution, a basic static website with JavaScript code could utilize the Google Maps API to showcase the various routes taken by the vans. This visualization would be beneficial for assessing the efficiency of a solution when adjusting the application's settings.

In addition, this functionality would be beneficial for the drivers of vans, as the platform could offer directions and scheduling assistance.

5.2.3 Expanding the Problem with full addresses

The current application operates based on orders that only include the ZIP code instead of the full address. However, in a real-life scenario, it would be necessary to extend the functionality to include real addresses.

This presents a challenge due to the large number of orders processed in a single day (approximately 2000 orders in the dataset provided by Enterprise-XY). Consequently, the size of the distance matrix would increase significantly. The dataset comprises around 730 unique ZIP codes, and if each order had a unique address, the number of unique locations would nearly triple (resulting in approximately 9 times the size of the matrix). In this section, we explore three approaches to address this issue.

The first approach is the simplest and most straightforward. Prior to routing, we generate a distance matrix that contains the addresses of the orders for a specific day. Currently, this distance matrix is pre-generated and consists of approximately 3200 unique ZIP codes. The intention behind this approach is to avoid regenerating the matrix each time the routing is performed. However, implementing this new method would require re-running it for every new dataset of orders. Currently, generating the distance matrix for all ZIP codes takes more than an hour and involves numerous API calls. If we were to regenerate it every time routing is executed, it would still take a considerable amount of time (although slightly less than before) and result in a significant number of API calls, which could incur costs. One advantage of this approach is that it would include information on road work and redirections in the distances and time for each day, which is not currently the case with the pregenerated matrix. Some modifications to the code regarding how the distance matrix is handled would also be necessary, but the impact on the code itself and performance would be minimal.

The second approach eliminates the need for a pre-generated distance matrix by directly accessing the Google Maps API for routing requests. However, this approach is slower than consulting an in-memory object, as API requests take tens of milliseconds, while the in-memory object takes nanoseconds. To mitigate this, we can cache the results of each query so that if the same query is repeated, we can use the cached response. However, this approach poses a challenge since the solver runs on a single thread, and each API request (which can be numerous) blocks the entire solver. This can significantly impact the quality of the solutions obtained within a reasonable timeframe. In the code, we would only need to modify the representation of locations and the implementation of the distance matrix. There are more changes required compared to the first method.

The third approach is relatively simple, but does not produce the most optimal results compared to the other two approaches. In this approach, the routing application remains unchanged, but after routing, the exact addresses are included in the solution. This means that the solver still plans the routes based on ZIP codes. However, the other two approaches utilise the exact addresses during routing, which allows for more precise results as the paths can be optimised to a greater extent. One major drawback of this approach is the use of the average distance or time between two locations within the same ZIP code. This can be problematic in situations where two orders are located at opposite ends of a ZIP code. In such cases, the planned time may not be sufficient to deliver both orders. To mitigate this issue, an alternative approach could be to use the maximum time between two ZIP codes instead (very pessimistic). This approach would probably work best in scenarios where there are many orders in small towns with few orders per ZIP code. However, it would be highly inefficient in large cities with a high concentration of orders within the same ZIP code.

In order to identify the most suitable approach among the three, it is necessary to conduct some testing. Ideally, if the generation of the distance matrix is efficient and if the program can handle a distance matrix of this size in memory, the first approach would be the optimal choice. However, if this is not feasible, but the second approach produces satisfactory results despite the API calls, then it would be the preferred option. If neither of these conditions are met, the third approach serves as a viable compromise as it builds upon a solver that has been proven to work and can deliver satisfactory outcomes.

5.3 Personal opinion

This project had various intriguing aspects. Firstly, it served as a practical implementation of constraint programming, which we learnt about in one of our courses. Additionally, it provided a valuable opportunity to familiarise ourselves with the Google OR-Tools library.

The library is extensive and complex, making it difficult to navigate. However, the guides provide a helpful introduction [1]. When I began using the library in my project, I needed more detailed information on its functionality and method usage. Unfortunately, the documentation for the code itself was inadequate. Consequently, I had to resort to examining the C++ source code [2] and consulting external resources that explained the library's inner workings [3] in order to comprehend how specific constraints or dimensions operated.

6 Honour Declaration

In this project, we used generative AI tools, namely Github Copilot for coding and Writefull for paraphrasing. Copilot was employed as an advanced autocomplete feature, but it did not generate a significant portion of the project. Writefull, on the other hand, was utilised to enhance the clarity of this document by employing its paraphrasing and grammar checking capabilities.

I, the undersigned Noah Godel, solemnly declare that the submitted work is the result of personal effort. I certify that I have not resorted to plagiarism or other forms of fraud. All sources of information used and author citations have been clearly acknowledged.

Acronyms

API Application Programming Interface. 22, 23

CSV Comma Separated Values. 3, 8, 10, 20

JSON JavaScript Object Notation. 3, 10, 11, 22

Glossary

Continuous Integration (CI) A software development practice involving automated and frequent integration of code changes into a shared repository. CI aims to detect and address integration issues early by incorporating automated build and test processes, ensuring a stable and reliable codebase. 17

Operational Research (OR) A discipline that employs mathematical and analytical methods to optimize decision-making processes in organizations. OR uses modeling and statistical techniques to provide data-driven insights for efficient resource allocation, logistics, planning, and decision support. 5

Vehicle Routing Problem (VRP) A optimization problem in the field operational research, where the goal is to optimize the routes of a fleet of vehicles to serve a set of locations, subject to various constraints. The objective is to minimize the total cost, which can include factors such as travel distance, time, or vehicle usage. 5

References

[1] Google. *Google OR-Tools Guides*. 2010. URL: `https://developers.google.com/optimization/introduction` (visited on 01/07/2024).

[2] Google. *Google OR-Tools Repository*. 2010. URL: `https://github.com/google/or-tools` (visited on 01/07/2024).

[3] Google. *or-tools User's Manual*. 2012. URL: `https://acrogenesis.com/or-tools/documentation/user_manual/manual/tsp/model_behind_scenes_overview.html` (visited on 01/17/2024).