Adding waypoint constraints to a shortest path routing algorithm in scheduled networks

Made in collaboration with Searoutes

Author : Martin Cornillad, L3 MPCI

Supervisor : Célia Châtell

November-December 2023

Table des matières

Acknowledgments

I would like to thank Célia Châtel, my internship supervisor, for offering my this opportunity of internship, for her involvement in it and her precious advice. I would also like to thank Pierre Garreau and Kelly Piriou for welcoming me in Searoutes, and allowing me to be a part of it for a month. Finally, I thank the whole Searoutes team for welcoming me warmly.

Introduction

Shipped merchandises represent 90% of the global merchandise trade. However, they're also responsible for the emission of over a 1000 Mt of CO_2 shared between over 55 000 ships. [1] The importance of cutting CO_2 emissions is permanently discussed, because of what impact it has on the planet. During the internship, I worked on routing algorithms designed by Searoutes. Those algorithms are the base of CO_2 emissions computing for companies, one of Searoutes area of activity.

The main purpose of this internship was to implement new functionalities in one of Searoutes routing application programming interfaces (APIs). This API computes fastest journeys between two points in the maritime network. The goal of the internship was to be able to force two new conditions on the computed journeys returned by the algorithm. The first one was to be able to force specific boats to be used, and the second one was to force specific locations to be used as a stop. The algorithm I worked with is inspired from the RAPTOR algorithm presented in the article Round-Based Public Transit Routing by Daniel Delling, Thomas Pajor and Renato F. Werneck [1]. After fully understanding what I needed to do, I started by studying the algorithm, then understood what modifications Searoutes did. To understand that, I studied the RAPTOR article and the Java implementation of the modified version of the algorithm. I talked a lot with the developer team of Searoutes in order to understand the logic behind the whole Java application. Then, I had to found a way to force specific conditions on the results of the algorithm and implement it inside the Java application.

The first section is a presentation of Searoutes, what activities are they doing and how are they doing it (page 2). The second section will focus on the already existing algorithm and implementation made by Searoutes. Notions needed to start modifying the algorithm and implementing other functionalities inside the application will be discussed here (page 3-9). The third section will be about how the algorithm was modified in order to implement specific conditions on the results, and how it has been built inside the Java application (page 10-15). In the last section, different tests will be detailed, and the performances of the API will be discussed (page 16-18).

Key Words : Algorithm, Routing, API, Java, Application

1. https ://www.polytechnique-insights.com/dossiers/energie/les-innovations-bas-carbone-du-fret-maritime/comment-reduire-lempreinte-carbone-du-fret-maritime/

1 Searoutes

1.1 Presentation of the company

Searoutes is a start-up founded in 2019 by Pierre Garreau, the current CEO. Being a full remote company, Searoutes has employees around the world, for a total of 12 people. However, they have an office in Zebox, a startup incubator from CMA-CGM in Marseille. This is where my internship was done. Startups working with Zebox are mainly in the "Transport, logistics and mobility" area of activity.

1.2 Searoutes activities

Searoutes develops application programming interfaces, also known as APIs. An API is a program that offers services to other programs through a specific interface. Searoutes sells access to those APIs to other companies. These are mainly used to compute C02 emissions and routes for shipping companies. There are routing APIs, CO2 APIs and vessel APIs.

CO2 APIs are mainly used to compute the carbon footprint of a company, over last years or upcoming years. Routing APIs can compute fastest itinerary between two points, but also the route distance and other information. This routing can be done in a direct way, on a geographic network, such as Google Maps, or on a scheduled or theoretical routing network. Finally, the vessel APIs give live information about the speed or position of vessels through the world.

1.3 Company organization

Pierre Garreau is the CEO of Searoutes. The 11 other employees are split between different fields. The main ones are the sales field, the tech field and the product field. Célia Châtel, my internship supervisor, is working as a VP of engineering at Searoutes, in the tech field.

2 Algorithm implemented by Searoutes

2.1 Explanation of the RAPTOR Algorithm

The algorithm used by Searoutes inside one of their API to compute best journeys in transit networks is inspired by the RAPTOR algorithm. Historically, most shortest path algorithms are based on Dijkstra algorithm. However, since the RAPTOR algorithm only deals with scheduled network, it uses a timetable instead of a graph. That's why it is a non Dijkstra-based algorithm. It computes all Pareto-optimal journeys (journeys that balance optimization equally between conditions) between two points. It needs to minimize arrival time and number of transfers. Let's consider the simple example of a bus system to explain the general idea behind the RAPTOR algorithm. A stop could be associated with a bus stop. A route would be a bus line, such as B3A in Marseille. A trip would be a bus going from the starting point of the route to the end point, at a given time in the day.
This algorithm works with rounds. For every round, it computes best arrival time at a stop by following every route route a maximum of once per round. During round number k, it is computing the fastest way to reach every stop in k trips. Every stop has a label, which is a set of values, named τ_0 up to τ_k (k being the maximum number of rounds the algorithm would be allowed to process).
There are four main steps during a round of the algorithm. These steps can be seen the pseudo-code of the algorithm 1. First, it associates to every stop its earliest known arrival time obtained from the previous round (τ_k is set to τ_{k-1} for every stop).
The second step is processing, for every route passing by at least a marked stop once, what is the first marked stop in the route (figure 1). Marked stops are stops which had their labels updated (and upgraded : $\tau_{k-1} > \tau_{k-2}$) in the previous round. When looking for the right stop on the route, it only consider marked stops. That is because stops that weren't marked in the previous round wouldn't be reachable during this round, since their best time weren't upgraded.

FIGURE 1 – Illustration of the choice made by step 2, on a route with marked stops in black and other stops in dark red

The third step is processing every route containing a marked stop. For every route, the algorithm traverses the route, starting from the specific marked stop found in the previous step. From this specific marked stop, it updates every stop label's after it if the new arrival time given by the route is better than the best arrival time at this stop (lines 13-20 in algorithm 1). If the stop label's happen to be upgraded, it is added to the marked stops for the next round. However, it may be possible that a quicker path to the stop is discovered, which means we may need to update the current ideal journey.
The fourth step would be computing the fact that you can actually walk between stops, which may help to get to some stops even faster. You can find a detailed explanation of the RAPTOR algorithm and its variants here [1]. The complexity of the algorithm is

$$O(K \times (R + T + F))$$

K being the number of rounds, R being the number of routes, T the number of trips and F the number of footpaths. The maximum number of rounds is set to 4. This makes it a very efficient way of computing fastest journeys between two points.

Two improvements can be made to the base algorithm, local and target pruning. Local pruning is adding another label to every stop, not round dependent, that only keeps the best label of the stop overall. This is useful because we only need to check if the best arrival time at the stop overall is improved. The RAPTOR algorithm finds the best journeys from one stop to every stop of the network. However, we only want the best journey to one specific Pt. We can then implement target pruning, which means we would consider a marked stop only if the arrival time stored in its label isn't higher than the label of the stop Pt, the destination stop.

Searoutes isn't using this exact algorithm. They have modified versions of the algorithm, depending on what conditions do they want to apply on the computed journeys.

Algorithm 1: RAPTOR Algorithm

Input : P_s the starting point, P_t the target point, τ the date of the start.
Parameters: K the number of rounds, T a set of trips
Output : A set of Pareto-optimal journeys

for $k \leftarrow 1$ **to** K **do**
  ; // Q is the list of marked stops
  clear(Q) ;
  ; // collect routes reaching marked stops (Step 2)
  foreach $p \in Q$ **do**
    foreach *route* r *serving* p **do**
      if $(r, p') \in Q$ *st* p *comes before* p' *inside* r **then**
        replace (r, p') by (r, p) in Q ;
      else
        add (r, p) to Q;
      end
    end
    remove p from *marked_stops*
  end
  ; // check if $\tau(p)$ can be improved (Step 3)
  for $(r, p) \in Q$ **do**
    $t \leftarrow \perp$ the current trip
    for $p_i \in r$ *after* p **do**
      ; // Can the label be improved in this round?
      if $t \neq \perp$ *used to reach* p_i *and* $arr(t, p_i) < \tau_k(p_i)$ **then**
        $\tau_k(p_i) \leftarrow arr(t, p_i)$;
        mark (p_i) ;
      end
      ; // Can we catch an earlier trip at p_i
      if $\tau_{k-1}(p_i) \leq \tau_{dep}(t, p_i)$ **then**
        $t \leftarrow$ earliest trip(r, p_i) ;
      end
    end
  end
  ; // looks at footpaths (Step 4)
  foreach $p \in Q$ **do**
    foreach $footpath(p, p') \in F$ **do**
      $\tau_k(p') \leftarrow min(\tau_k(p'), \tau_k(p) + l(p, p'))$
    end
  end
  if *marked_stops is empty* **then**
    break
  end
end
Return ReconstructJourney()

2.2 Modified version of the algorithm used by Searoutes

Let's consider the actual algorithm used by Searoutes. It is different from the actual RAPTOR algorithm proposed in the research paper. The first main difference is that boats are following infinite rotations on a network, which means they doesn't have a starting point, or an ending point. This means there is no difference between routes and trips, they are only considering trips. The second difference is the fact that we don't have

to compute footpaths here. Searoutes only needs to consider full boat trips for this API.

There is no reconstruction of the journeys inside the base algorithm because it only computes best arrival times, so it also needs to be added. In order to build the journey, every time a label is updated, the stop giving the new best arrival time is saved. It is called the parent. This way, the journey can easily be build back when it's fully computed.

Searoutes called this version of the algorithm the ForwardRaptorAlgorithm (algorithm 2). However, this version does not take into consideration how long the returned journeys are (time wise), i.e. when does the trip start. Getting the trip that starts last is important, because carriers ship every week most of the time, so they do not care when the trip starts, but they want it to be the fastest possible. That's why another variant of the algorithm is needed, called the BackwardRaptorAlgorithm. It computes the journey that start the latest in all the journeys returned by the ForwardRaptorAlgorithm. These two versions of the algorithm are used together, forward version computes the journeys and backward version returns the ones starting the latest in the forward computed journeys.

Algorithm 2: ForwardRaptorAlgorithm

Input : P_s the starting point, P_t the target point, τ the date of the start.
Parameters: K the number of rounds, T a set of trips
Output : A set of Pareto-optimal journeys

```
for k ← 1 to K do
    clear(Q) ;
    ; // collect trips reaching marked stops
    for p inside marked_stops do
        for t inside T serving P do
            if p' ∈ marked_stops st (t,p') exists and p comes before p' inside t then
                replace (t,p') by (t,p) in Q ;
            else
                add (t,p) to Q;
            end
        end
        remove p from marked_stops
    end
    ; // check if τ(p) can be improved
    for (t,p) ∈ Q do
        for p_i ∈ t after p do
            if arr(t,p_i) < τ*(p_i) and arr(t,p_i) < τ*(p_t) then
                τ_k(p_i) ← arr(t,p_i) ;
                τ*(p_i) ← arr(t,p_i) ;
                mark (p_i) ;
            end
        end
    end
    if marked_stops is empty then
        break
    end
end
Return ReconstructJourney()
```

2.3 How is it implemented in the app

During the internship, I studied a part the full structure of the application used by Searoutes. The application that is running the algorithm that I worked on is using Spring. Spring is a Java framework that allows developers to focus on the application, while it is handling the infrastructure. I didn't study how does Spring works, but I had to use some of its functionality, especially for running tests. This whole application

is a part of the network of APIs built by Searoutes.

2.3.1 General structure

The Routing Controller is the entry point of the application. It is a class that handles the parameters given by the user to ensure that they're valid, give the order to run the parts dedicated to the right algorithm, and give back the results. It also handles error cases. Everything related to the algorithm is made thanks to the class Routing Service. It has many methods, used to compute journeys thanks to the forward Raptor algorithm, or the backward version, or both. Routing Controller calls the RoutingService.getJourneys method. This method calls other methods depending on what are the parameters given by the user (for example if the user only wants journeys starting in a specific time range). All those methods set up how the algorithm is going to be run, if there is a specific time range, then we need to modify the timetable. The timetable is a class that holds all the trips available to be used by the journeys. Eventually, all those methods will call the runRaptor method, that runs the algorithm, and returns a set of journeys. Those journeys will go through other methods of Routing Service in order to format them. Then, the Routing Controller get them back, and displays them to the user.

2.3.2 Implementation of the algorithm

Every class used to implement the algorithm are displayed in the UML diagram in the annex in figure 7. The ForwardRaptorAlgorithm and the BackwardRaptorAlgorithm are classes inheriting from an abstract class AbstractRaptorAlgorithm. An abstract class is a class containing abstract methods, methods that are declared but aren't implemented such as the following examples. (listing 1)

Listing 1 – Examples of abstract and implemented methods

```
1  //Implemented method
2  public void reset() {
3      bestLabels = new LabelBag();
4      bestLabelsOfRound = new ArrayList<>();
5      markedStops = new HashSet <>();
6      currentRound = 0;
7          }
8
9  //Abstract method
10     protected abstract RaptorResult reconstructJourney(RaptorSearchParameters
           raptorSearchParameters);
```

Abstract methods are used because Forward and Backward algorithm have similar structures, but they're some slight differences inside specific parts. The Abstract Raptor Algorithm is used to set up the structures of both algorithms and then every method specific to the algorithm is define inside its own class. There are only two methods implemented inside the Abstract Raptor Algorithm. The first one is the method used to reset the parameters of the algorithm, such as the labels. The other one is the run method, that calls every method in order to run the algorithm. (listing 2)

Listing 2 – Run method of the Abstract Raptor Algorithm class

```
14 public RaptorResult run(RaptorSearchParameters raptorSearchParameters) {
15     //Initialisation
16     this.reset();
17     this.initBags(raptorSearchParameters.nMaxRounds, raptorSearchParameters.
           searchSource,
18     raptorSearchParameters.startSearchDate);
19
20     for (int k = 1; k <= raptorSearchParameters.nMaxRounds; k++) {
21         currentRound = k;
22         if (!markedStops.isEmpty()) {
```

```
23          //Step 2
24              Map<Trip, StopTime> routeStops = this.accumulateRoutes(
                    raptorSearchParameters);
25          //Step 3
26              this.traverseRoutes(routeStops, raptorSearchParameters);
27          }
28      }
29      return reconstructJourney(raptorSearchParameters);
30  }
```

Steps defined in the section Explanation of the Raptor Algorithm can be seen here. Step one is hidden in the accumulatesRoutes method. Step 2 and 3 are the accumulateRoutes and the traverseRoutes methods. Step 4 isn't implemented because it is useless as explained earlier.

2.4 Forcing an IMO inside the journeys

2.4.1 Theory

From here, we will refer to a combination of loading on a vessel, traveling on it, then unloading as a leg of a journey. The IMO number is a number used to identify boats. Every boat has its own IMO, used to refer to them.

Forcing an IMO means that the journeys returned by the algorithm need to have at least a leg that is using this IMO. To be sure that we only get solutions with this IMO used in at least one leg, we can create a loop that runs a slightly modified version of the algorithm multiple times. This modification does not modify how the algorithm works without any forced IMO, since the modified part of the algorithm only activates if there is a forced IMO. This loop, lets call it the outside loop, iterates a value, called i, from 1 to the maximum number of rounds we allow our algorithm to make.

Every iteration of this loop, the algorithm will return journeys where the leg number i uses the forced IMO, and every other leg doesn't use the forced IMO. Other legs can't use the forced IMO, otherwise we would get unusable answers, such as unloading and loading again on the forced IMO in the same stop, but with a multiple days long interval. This is because when a container is unloaded from the boat, it needs to wait a certain amount of time, called transshipment time, before loading again on another boat. Let's consider this simple example. Let's say that the specific IMO is forced on the second leg. However, it's also the best IMO to use in the first leg. The computed journey in this iteration of the loop would look like the journey represented in figure 2.

We stop the loop if the length (in legs) of the shortest journey computed yet + 2 is lower than the number of the leg where we are currently forcing the IMO. The upper limit of the outside loop, which is the number of rounds, is given by the fact that a journey will have a maximum number of legs equals the number of rounds, so forcing the IMO on a leg whose number is higher than the number of rounds doesn't make sense.

The lower limit of the loop is a heuristic. This means the optimal condition on the answer is lost to benefit the speed of the algorithm. We cut the search for the best answer earlier considering that : When the number of the leg where the IMO is forced is higher than the number of legs in the shortest answer we found + 2, the loop can be stopped. This is because carriers do not care for small improvements in time, if it is thanks to a bigger amount of legs. That's because during a transshipment, containers could potentially missed the next vessel because of a longer transshipment time than expected.

The modified part in the RAPTOR algorithm is inside the loop that goes from 1 to the number of rounds allowed (starting at line 3 in algorithm 1) when you consider for every marked stop, all trips passing by this stop. If the iteration number (the round the algorithm is in) matches the number of the leg we want to force the IMO on (called i earlier), we only consider trips using this IMO. Otherwise, we consider every trip passing by the marked stop, except the ones using the forced IMO . The n-th iteration of the loop inside of the algorithm is responsible for the creation of the leg number n. Only considering trips using the forced IMO on the n-th iteration means that if the n-th leg is updated, it will use the forced IMO.

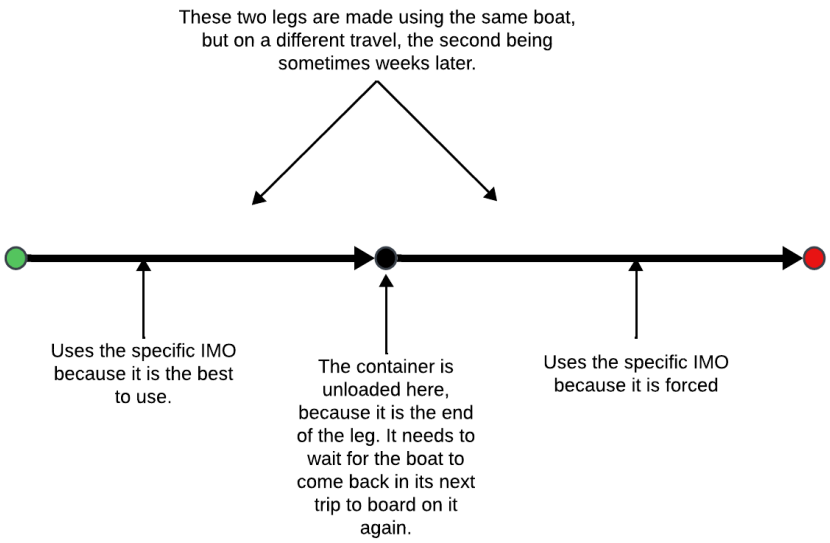

FIGURE 2 – Illustration of the unusable answers if every leg can use the specific IMO

2.4.2 Implementation

The implementation for the single forced IMO was already done. Here is a brief explanation :

The Via class is used to store the IMO we want to force and the leg we want to force it on, and holds the methods to interact with the algorithm. The two main methods are : the builder of the class and IsCompatible() (Listing 3). The builder takes the forced IMO and the number of rounds as parameters and returns two maps (a map is a set of element, each element is a pair of key and value, keys are unique), IMOPerLeg and blockdImoPerLeg. IMOPerLeg contains one association : a number of rounds between 1 and the maximum number of rounds associated with the forced IMO. blockedImoPerLeg contains every other association, any number of rounds that is not in IMOPerLeg associated with the forced IMO. Here is a little visual example :

IMOPerLeg		<2, 9619907>	
blockedImoPerLeg	<1, 9619907>		<3, 9619907>

This Via is the parameter given to the algorithm. It does change every call made to the algorithm. If IMOPerLeg is <1, IMO>, the IMO would be forced on the first round and blocked on every other round. This means, by creating a Via for every round, and iterate on those, we can generate the outside loop, and properly run the ForwardRaptorAlgorithm. The map IMOPerLeg could be enough to properly run the loop. We only need IMOPerLeg to check if this is the round were the IMO is forced or not. However, the main goal behind implementing new functionalities in the application is too stay as general as possible. Having a map that contains blocked IMOs per round creates more possibilities. A new condition could be created, where a specific IMO is forced, but an other IMO is totally blocked from being used. That is why there is two maps. The method isCompatible() is build in the same idea. It is used to test if a trip is compatible with the forced IMO or not, depending on the round. That's why the parameters are the trip we want to check, and the round the algorithm is currently in. The condition on line 7 of algorithm 4 in the annex (or line 56 in the listing 3) is true if isCompatible() returns true, and false if it returns false.

Listing 3 – Java code for the Via class

```
31 public class Via {
32
33     Map<Integer, Integer> IMOPerLeg = new HashMap<>();
34     Map<Integer, Set<Integer>> blockedImosPerLeg = new HashMap<>();
```

```
35
36      public Via(Map<Integer, Integer> IMOPerLeg, int maxRounds) {
37          this.IMOPerLeg = new HashMap<>(IMOPerLeg);
38          for (int i = 1; i <= maxRounds; i++) {
39              blockedImosPerLeg.put(i, new HashSet<>());
40          }
41          for (Map.Entry<Integer, Integer> legNumberAndImo : IMOPerLeg.entrySet()) {
42              for (int i = 1; i <= maxRounds; i++) {
43                  if (i != legNumberAndImo.getKey()) {
44                      blockedImosPerLeg.get(i).add(legNumberAndImo.getValue());
45                  }
46              }
47          }
48      }
49
50      public boolean isCompatible(int currentRound, int IMO) {
51          Integer IMOOfLeg = IMOPerLeg.get(currentRound);
52          Set<Integer> blockedImosOfLeg = blockedImosPerLeg.get(currentRound);
53          if (blockedImosOfLeg == null && IMOOfLeg == null) {
54              return true;
55          }
56          return (IMOOfLeg != null && IMO == IMOOfLeg) || (IMOOfLeg == null && !
                blockedImosOfLeg.contains(IMO));
57      }
58  }
```

3 Implementation of the new conditions

As said earlier, the end goal of the internship was to implement new conditions on the computed journeys returned by the Searoutes algorithm. The initial condition was to be able to force multiple specific IMOs to be used. This can be seen as an expansion of the condition discussed in section 2.4. The second condition was to be able to force the journey to have one or multiple specific locations as stops. The user gives a list of specific locations, and the computed journeys needs to have those locations as stops. The end goal of the internship was to implement those two modifications inside Searoutes application and to be able to force the two conditions at the same time.

3.1 Theory

3.1.1 Multiple forced IMOs

To force multiple IMOs to be used inside the computed journeys, the journeys need to have at least each IMO used once in one of the legs. In order to force the algorithm to compute such journeys, we can use the same idea that in section 2.4. However, forcing one IMO on a specific leg for every iteration of the loop wouldn't work. For every iteration of the loop, we need to force a combination of all the forced IMOs to be used on a combination of legs. Let's consider an example. If the journeys can have a maximum of three legs, and we want to force two IMOs on the computed journeys, this would be every combination of IMOs and legs that would need to be tested :

ITERATION Nb	IMO forced on LEG 1	IMO forced on LEG 2	IMO forced on LEG 3
Iteration 1	imo_1	imo_2	none
Iteration 2	imo_2	imo_1	none
Iteration 3	imo_1	none	imo_2
Iteration 4	imo_2	none	imo_1
Iteration 5	none	imo_1	imo_2
Iteration 6	none	imo_2	imo_1

Let's say the number of legs is n and the number of forced IMOs is p. The number of combinations that needs to be tested is the number of arrangements of p values in n possibilities. The maximum number of iterations that can be done in the outside loop is 24, with n=4, the maximum number of legs, and p=3 or p=4 the maximum number of IMO forced. This could have an impact on the performance of the application, this will be discussed in section 4. Once again, if an IMO is forced on a specific leg, every other leg in this journey can't used this IMO. The lower limit of the loop is also a heuristic. The loop is stopped only if the minimum number of round to respect the condition (i.e. the number of the last leg where an IMO is forced) is higher than the length of the shortest answer found yet + 2. This is an acceptable condition to stop for the same reason than in section 2.4, companies would rather have decently fast answers with few legs than the shortest answer (time wise) but with a lot of legs. The upper limit of the loop is reached when every arrangement has been tested.

3.1.2 Mutiple forced locations

In this section, forcing one location or multiple locations is done in the same way, so there is only a general explanation. To ensure that the computed journeys uses the forced locations as stops, every location needs to be forced during a round of the algorithm (i.e. on a leg of the journey). Forcing a location to be used during a round means that during the round, this location have to be the only marked stop. This ensures that the computed journey have to use this location as the end of the leg of the round. This also means that the leg computed in the next step will also use this location as its start. The modification in the algorithm is made in the line 14 (algorithm 2). Instead of allowing every stop located after the stop stored in (t,p) to have their label updated, there is a condition. If it is not a round where a location is forced,then any p_i's label can be updated, except if p_i is one of the forced location. If it is a round where a location is forced, then the only stop that is allowed to be have its label updated is the one corresponding to the forced location. Once again, blocking the forced locations to be used on a leg where they are not currently forced is needed.

Otherwise, unusable answers would be computed. For example, if an execution allows a 4-leg journey with a forced location on the third round (i.e. in the end of the third leg), we could get the answer from figure 3.

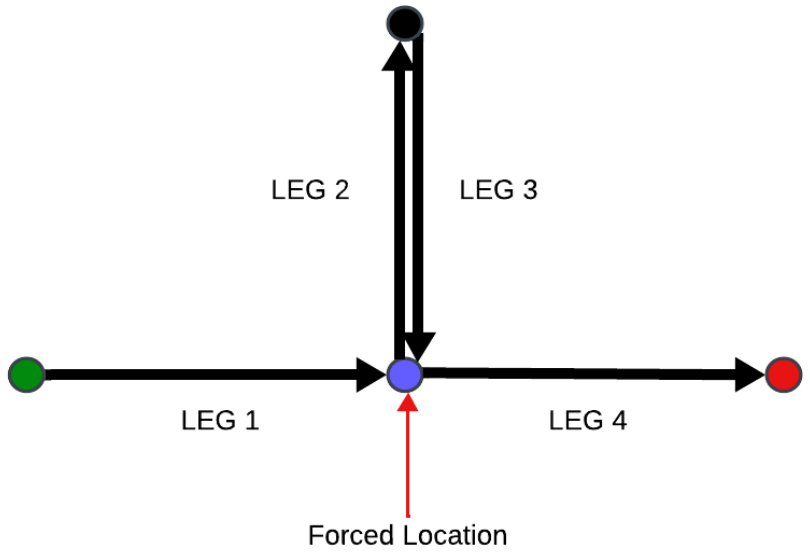

FIGURE 3 – Illustration of the journey that could happen if the "blocking" condition isn't used

This would happen if the leg computed in the first leg uses the forced location in the end. However, with the "blocking" condition, this would be avoided. As earlier, the algorithm is run by a loop that iterates over every arrangement of the forced locations in the different legs. The maximum number of forced locations is three, which means the maximum number of time the algorithm is ran is 6, because $\frac{3!}{(4-3)!} = 6$ $(0! = 1!)$ (Only 3 legs possible, for 3 locations). The lower limit of the loop is set by the same heuristic. The loops stops if the number of legs + 2 in the shortest journey computed yet is lower than the minimum number of legs a journey needs to have to force every location during this iteration. The pseudo-code for this part can be found in algorithm 5 in the annex.

3.1.3 Forcing both at the same time

In order to force both IMOs and locations at the same time, the algorithm is run over every arrangement of IMOs associated to round and every locations associated to round. This is done by generating a list that holds every arrangement of the forced IMOs and locations associated to rounds. The stop limit for the loop is still the same. The lower one is if the number of legs of the shortest computed journey yet + 2 is lower than the minimum number of leg to force every condition. The upper limit is reached if every arrangement has been used by the algorithm.

3.2 Implementation

In this part, every location is also a locode, which is a five letter code used to describe locations easily. For example, in France, Le Havre's port is called FRLEH. Locode is also the name of the Java type used to hold the location's information.

The already existing Via class wasn't built in a way that would allow a proper implementation of the new conditions. The Via class could have been modify to implement the two new conditions. However, this would have been a very specific implementation, that wouldn't allow other conditions to be implemented in the future. The goal was to implement a new Via class, that would be able to accept new conditions with a

minimal amount of modifications.

The idea behind the new Via class was to implement a design pattern, called a composite structure. "Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code. Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects." [3]. for the implementation of the new conditions, the tree is composed of a node and two leaves. The node is called the via composite, and the two leaves will represent each conditions, forced IMO and location. Here is a general idea of the composite. There are three different classes/interfaces :

1. The component interface : An Java interface is used to define the behavior of a Java class and its methods but without implementing it. The composite interface is used to define the general behavior of every elements of the tree.
2. The composite class : This class could be compared to a container. It doesn't hold much logic, and doesn't know what are the concrete classes of its children. In the tree, this is a node. It delegates all the work to its children. It implements the interface.
3. The leaf classes : Every leaf of the tree can have its own classes. However, since they also implement the interface, they need to implement the methods that are in the component. These classes hold the logic used to answer when their composite are asking information through methods.

I chose to use this design pattern to make a general implementation of the via. Once again, having the most general methods and structures is essential. This structure will be able to receive new forced conditions by adding new leafs or composites, the structure will not need any modifications. That's because the logic behind every condition is implemented in the leaves, and the composite is just making requests to those leaves. Adding leaves can be done easily.

The component interface is called the ViaInterface. It defines 6 different methods that will be shared and implemented both by the composite and the leaves. These methods are :

1. boolean isCompatibleAccumulateStep(int currentRound, Trip trip) : This method will be used to check if the trip picked in line 6 (algorithm 4) can be used, considering every condition forced on the computed journeys. In the implementation of the forced IMOs or locations, the only condition that will be checked is the one on line 7 (algorithm 4).
2. boolean isCompatibleTraverseStep(int currentRound, StopTime stopTime) : It works in the same way than isCompatibleAccumulateStep() but on the stops picked in line 15 in algorithm 5.
3. boolean hasViaAtRound(int roundIndex) : This methods returns a boolean, true if the round taken as a parameter has a condition forced on it, false if it doesn't have any forced conditions.
4. int minNRoundsToRespectVia() : Returns the last round that has a condition forced on it, i.e. the minimum number of leg the journey needs to have in order to be able to respect every condition.
5. boolean hasVia() : returns true if there is at least one condition forced on the computed journeys.
6. ViaInterface mirrorVia(int maxNbRounds) : This methods is used to generate the equivalent of the via used in the forward algorithm, but for the backward algorithm. That's because we want the backward algorithm to search for solutions that has conditions in the same legs. Since it computes the journeys starting from the end, it needs to have a different via, i.e. a mirrored via.

The composite implements each of those methods, in a very simple way. The only method implemented by the composite that is not defined inside the interface is its builder. It takes a list of Via interface as a parameter and sets a list called components to this list, as shown in the listing 4.

Listing 4 – Java code for the Via interface

```
59      public ViaComposite(List<ViaInterface> componentsToAdd) {
60          this.components.addAll(componentsToAdd);
```

This list contains the leaves used by the Via composite to compute the results of every methods. In this situation, only two leaves are needed. One for the forced IMO condition and one for the forced location condition. These two leaves also implement the Via interface. They hold the logic that helps the composite to return the result for every method. The detailed code of every classes can be found in the annex in listings 7, 8, 9, 10 in annex and the behavior of every method is described in table 1.

Methods	Composite	IMO leaf	Location leaf
`boolean isCompatible AccumulateStep`	Returns True if the method returns True for every x in X	Returns True if the trip is compatible with the forced IMOs depending on the round (condition line 7 on algorithm 4)	Returns True (The forced locations don't matter when verifying if a trip is compatible with the forced IMO)
`boolean isCompatible TraverseStep`	Returns True if the method returns True for every x in X	Returns True (The forced IMOs don't matter when verifying if a stop is compatible)	Returns True if the stop is compatible with the forced locations depending on the round(condition line 15 on algorithm 5)
`boolean hasViaAtRound`	Returns True if the method returns True for at least one x in X	Returns True if an IMO is forced in the round taken as a parameter	Returns True if a location is forced in the round taken as a parameter
`int minNRoundsTo RespectVia`	Returns the maximum between the values returned by the method for every x in X	Returns the number of the last round where an IMO is forced	Returns the number of the last where a location is forced
`boolean hasVia`	Returns Returns True if the method returns True for at least one x in X	Returns True if at least one IMO is forced	Returns True if at least one location is forced
`ViaInterface mirrorVia(int maxNbRounds)`	Returns a Via Interface with the results of the mirrorVia methods from every x in X	Returns another Via IMO leaf but with the forced IMOs adapted to the backward version of the algorithm (To force the same condition when computing backward)	Returns another Via location leaf but with the forced locations adapted to the backward version of the algorithm (To force the same condition when computing backward)

TABLE 1 – Table of the logic of the different Via classes, X is the list of components, x is a component

For the IMO leaf, the implementation is very similar to the one in section 2.4.2. There is a map of rounds associated to IMOs called forcedImoPerLeg. The builder has to create the blockedImoPerLeg map again, to stay as general as possible. However, blockedImoPerLeg's type is now Map< Integer, Set<Integer> > because there is more than one IMO blocked per leg if we force more than one IMO. This works the same way for the location leaf, where there is a forcedLocationPerLeg map, that associates rounds to forced locations, and a map called blockedLocationPerLeg. Once again, the blocked list is added in order to easily modify the code in the future, to block some locations, even if they are not forced.

However, to properly compute the journeys, we need to run the algorithm multiple times on every arrangements of forced IMOs and forced locations associated to rounds. This is done thanks to a method called getViaToTest, that takes two sets, one of IMOs and one of locations, and an integer, the number of rounds. This method returns a list called vias. This list holds every Via composite possible, that have in their component list, one of the arrangement of IMOs and locations with rounds, stored inside the via IMO and locations leaves. Here is a simple example, where the number of rounds is set to 2, and there are forced locations, FRLEH and NOOSL, and a forced IMO, 9619907. The list returned by the method getViaToTest would hold 4 Via composites, built as shown in table 2.

To compute every arrangement of IMOs with rounds, and every arrangement of locations with rounds, I made a method called generateAllArrangements. This methods takes two lists as parameters, a list with

Elements of the list vias	Conditions forced with the IMO leaf	Conditions forced with the Location leaf
Via Composite 1	9619907 on L_1	FRLEH on L_1 and NOOSL on L_2
Via Composite 2	9619907 forced on L_1	NOOSL on L_1 and FRLEH on L_2
Via Composite 3	9619907 on L_2	FRLEH on L_1 and NOOSL on L_2
Via Composite 4	9619907 on L_2	NOOSL on L_1 and FRLEH on L_2

TABLE 2 – Table of the example detailed above, L_i represents leg number i

every round in it, and a list with every forced IMOs (or locations), and an index (used for recursive calls). It returns a list containing a list of every forcedImoPerLeg(or locationPerLeg) possible, to generate the leaves. These leaves are then used to generate all the Via composite, hold by the vias list. This method returns a type that is List< Map<T, U> >. T and U are generic types used to be able to define the function without specifying the types of the elements holds by the maps. This is useful because if there is another place in the code where arrangements need to be computed, this same method can be used, without modifying it if the types inside the maps aren't the same. This is a recursive function. Here is the pseudo-code, in algorithm 3. Its implementation can be found in the annex in the listing 11.

Algorithm 3: Recursive **generateAllArrangements**

Input : *keys*, *values* two lists of integers with number of keys ≥ number of values, i the current index
Output: A List containing every map associated to every arrangement of keys and values possible (order and no repetition)

```
1  updatedList = Empty List
2  if i = size of (values) − 1 then
       ; // Stop case
3      foreach key ∈ keys do
4          map = empty map
5          add (key, values.i) in map
6          add map in updatedList
7      end
8  else
       ; // Recursive case
9      currentList = generateAllArrangements(keys, values, i + 1)
10     foreach map ∈ currentList do
11         foreach key ∈ keys do
12             if map.key is null then
13                 temporaryMap = empty map
14                 add (round, values.i) in temporaryMap
15                 add temporaryMap in updatedList
16             end
17         end
18     end
19 end
20 Return (updatedList)
```

Every recursive call, the method takes the list of map takes the list of maps already made, then pick an IMO (which the element in values associated to the index). Then, for every map in the already existing list, it generate every new map possible, by adding the selected IMO at every round possible, where an IMO wasn't already forced. This is a little example, considering that there is three IMOs in the list, and three rounds possible, in figure 4.

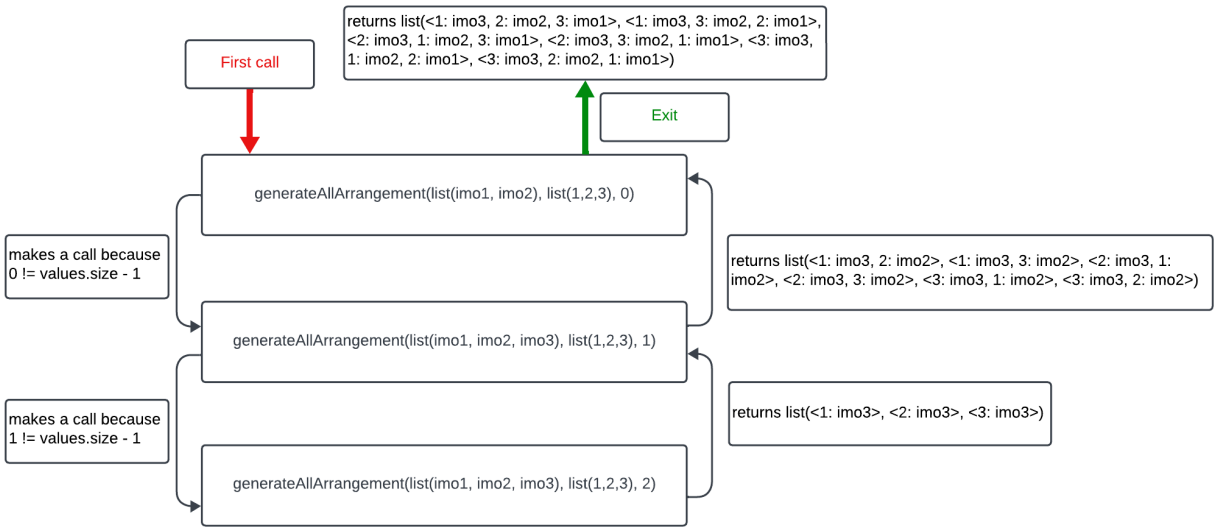

FIGURE 4 – Example of the behaviour of the method generateAllArrangement with 3 IMOs and 3 rounds

Finally, let's take a look at the runRaptor method. As said earlier, this is the method used to iterate multiple times over the algorithm. This is the method used to compute journeys. The java code of the method can be found in listing 5.

Listing 5 – Java code for the runRaptor method

```
61      public static List<RaptorResult> runRaptor(RoutingSearchParams routingSearchParams,
62              AbstractRaptorAlgorithm raptorAlgorithm,
63              RaptorSearchParameters basicRaptorSearchParams) {
64          List<RaptorResult> results = new ArrayList<>();
65          int maxNbRounds = routingSearchParams.getMaxNbRounds();
66          int nbLegsMin = maxNbRounds;
67          RaptorSearchParameters viaRaptorSearchParameters;
68          for (ViaInterface via : routingSearchParams.getViaToTest()) {
69              // Do not try to find solutions with more than 2 more legs compared to the
                    solution
70              // with the least legs. Meant to reduce the number of searches but might
                    hide good
71              // solutions
72              int maxNbRoundsAllowed = Math.min(maxNbRounds, nbLegsMin + 2);
73              if (via.minNRoundsToRespectVia() <= maxNbRoundsAllowed) {
74                  viaRaptorSearchParameters = RaptorSearchParameters.copy(
                        basicRaptorSearchParams);
75                  viaRaptorSearchParameters.setNMaxRounds(maxNbRoundsAllowed);
76                  viaRaptorSearchParameters.setVia(via);
77                  RaptorResult raptorResults =
78                          raptorAlgorithm.run(viaRaptorSearchParameters);
79                  nbLegsMin =
80                          Math.min(
81                                  raptorResults.getResults().stream().map(j -> j.getLegs
                                        ().size()).min(Integer::compareTo)
82                                          .orElse(nbLegsMin), nbLegsMin);
83                  results.add(raptorResults);
84              }
85          }
```

This method takes parameters for the journey, such as starting point, destination or the starting date for the search of the journeys. It returns a list of elements, of type RaptorResult, which is a type used to store a journey computed by the algorithm, but not formatted yet to be displayed to the user. This method iterates over the elements stored in the vias list, the list returned by the method getViaToTest(). For each via Composite stored in it, if the minimum number of leg the journey needs to have to be able to respect every condition stored in the Via composite is lower than the length of the shortest journey computed yet + 2, it computes the journeys using this Via composite. This means it considers a Via composite only if the lower condition of the loop is not reached yet, the same condition as section 3.1. Then, the computed journeys are stored in the list, to later be formatted and displayed to the user.

4 Testing functionalities and performances

In order to test the newly implemented functionalities, there are two types of tests that were made. First, tests that verified that the new classes and methods were behaving properly. Then, tests that verified the impact of the new functionalities on the performances of the API.

4.1 Testing the new functionalities

To actually write a test method, the @Test was used, it comes from the JUnit library. This is an annotation that allows us to specify that the method written right after this annotation is meant to be a test. Test methods return a void type. The java code of every test written during the internship can be found in the annex, in listings 13, 14 and 15.

4.1.1 Unit tests

Unit tests are used to verify the behavior of a single method. By adding tests every time a method is created or modified, we can make sure that the method behave properly. This means that if the tests are run every time a modification is changed, bugs can be detected and patched before the modifications are released to the clients.

Using a simple network, composed of 4 locations and 7 trips, we're able to test the functionality of the forward algorithm (and the backward), to make sure the new conditions implemented in the algorithm works properly. This is done by creating a Via composite manually, and giving it to the algorithm, getting the results back, and then verifying that the conditions are properly forced. An example of a test made to verify that both IMOs and locations could be forced at the same time can be found in listing 12 in annex. In line 342, we check that the number of journeys returned is equal to the expected number, which is 1. Then we verify that the characteristics of the returned journey matches the journey we were expecting. This means, starting and end point are verified, the number of legs, the IMOs and locations used too. This allows us to verify that every condition were respected, and that the returned journeys aren't broken or bugged. If the Via composite structure is modified in the future, running this tests would allow the developers to verify that the Via composite still works for IMO and location. This is just an example, but every possibility is tested. This includes no solution situations and unusual queries. For example, I also added tests were the algorithm, because of the forced conditions, needed to pick the worst possible answer without conditions.

The method generateAllArrangement also needed to be tested. Since it is a very generic method that could be used somewhere else in the code than its first purpose, it was tested on a generic set of elements, to make sure it was working on any type of elements, and not the specific ones used in the generation of the Via composites. To test it, the expected list is built by hand, and then compared to the actual result of the function, to make sure they are the same.

4.1.2 Integration tests

Integration tests are meant to test the behavior of the entire application. This means that these methods make real requests to the API, get the answers back and check if they're correct. To be able to make queries and check if the answers are right, the queries and answers are transformed into a JSON format, which stands for JavaScript Object Notation. This is a textual data format. It is very useful because it allows us to get

precise information about the answers. They are two types of integration tests. Some tests are testing if the answers are correct. For example, if we forced an IMO to be used, it checks if every answer computed has that IMO. The other ones are testing error cases. The error code is analyzed to make sure every information is properly displayed. This is the test method that checks when a wrong IMO format is used (listing 6) :

Listing 6 – Test method for an error case where the IMO used is in a wrong format

```
86      @Test
87      public void simple_route_wrong_wrong_imo_format() {
88              String json =
89                      given().
90                              param("fromLocode", "CNSHA").
91                              param("toLocode", "CNSHA").
92                              param("fromDate", "2023-03-22").
93                              param("viaImos", "eeee").
94                              when().
95                              get("/itinerary/v2/route").
96                              then().
97                              statusCode(400).
98                              extract().asString();
99
100             ReadContext jsonPath = JsonPath.parse(json);
101
102             assertThat(jsonPath.read("$.errorCodes").toString(), equalTo("[\"5000\"]"))
                    ;
103             assertThat(jsonPath.read("$.messages").toString(), equalTo("[\"Error when
                    parsing " +
104                     "'viaImos' as an integer.\"]"));
105     }
```

In line 102, the method checks if the error code is the right one. Error code are used to give information about what type of error happened. In line 103, it verifies that the error code is the expected one.

4.2 Testing Performances

To test performances, a python script was used. Thanks to a list of the locodes of the 200 biggest locations, 400 different pairs of starting and ending points were generated. For every pair, 12 requests were made. One with no via, 4 with one, two, three and four random IMOs forced, 3 with one, two and three locations forced and 4 with one IMO and one location forced, one with two IMOs and one locations, one with one IMO and two locations and finally one with two IMOs and two locations. The answers given by the API were stored in a .CSV file. An example of the data stored in the .CSV file for a pair of starting and ending point is shown in figure 5.

	code	response_time	from	to	search_date	via_imos	via_locations	duration	nb_transhipments
133	200	2.18041	NOOSL	AUADL	2023-06-01	[]	[]	2.24	2
134	200	1.31856	NOOSL	AUADL	2023-06-01	9437799	[]	12.06	4
135	404	0.59734	NOOSL	AUADL	2023-06-01	9501710,9542893	[]	none	0
136	404	0.25466	NOOSL	AUADL	2023-06-01	9721633,9326706,7117979	[]	none	0
137	404	0.15636	NOOSL	AUADL	2023-06-01	9303807,9780665,9312717,9612997	[]	none	0
138	200	3.46731	NOOSL	AUADL	2023-06-01	[]	SEGOT	0.53	3
139	200	0.92124	NOOSL	AUADL	2023-06-01	[]	USBOS,MYPKG	2.7	4
140	404	0.14697	NOOSL	AUADL	2023-06-01	[]	GBABD,ARBUE,AUSYD	none	0
141	404	1.07357	NOOSL	AUADL	2023-06-01	9348493	GBSOU	none	0
142	404	0.9139	NOOSL	AUADL	2023-06-01	9815331,9445588	USJAX	none	0
143	404	0.619	NOOSL	AUADL	2023-06-01	9144251	CNLYG,TRIZM	none	0

FIGURE 5 – Example of the data stored in the .CSV file for a pair of locode

If the code is 200, this means the answer to the query is at least one journey. If it is 404, this means the query was correct, but no answer were found, while respecting the forced conditions. Doing random queries isn't a very efficient way of generating data for performances evaluations. That is because the proportion of

answers with a code 200 is only 27 percents, and code 404 is 73 percents. This means a lot of the data is not really representative of the true performances of the algorithm. When clients are making requests to the API, they are not random, which means most of the time, they would get a code 200. Unfortunately, using random requests was the only option available.

The average response time for answers with code 200 is 1.886 seconds, and for code 404, it is 0.729 seconds. This shows that answers that doesn't return a journey are on average faster than the one giving a journey back. This is because, when the forced conditions doesn't make sense, the algorithm could never be able to start a journey, which means every iteration would end really fast. For example, if we are forcing an IMO referring to a boat only used in Asia, but the journey is between to American cities, every iteration would end in the first round.

Without any forced conditions, the average response time of a query is 1.296 seconds. The table in figure 6 allows us to compare the average response time difference between no condition forced and every type of conditions possible when the code is 200, except for 3 and 4 IMOs forced, or 2 IMOs and 2 locations, because the amount of code 200 was too low.

	n_via_imos	n_via_locations	response_time	response_time_vs_no_via
0	0	0	1.30	0.00
1	0	1	2.56	1.24
2	0	2	1.83	0.60
3	0	3	0.47	-0.46
4	1	0	2.00	0.70
5	1	1	2.18	0.93
6	1	2	2.15	0.72
7	2	0	1.42	0.30
8	2	1	1.47	-0.20

FIGURE 6 – Table that shows the average response time difference with no condition for every condition

The biggest time difference (on average) is obtained by forcing only one location. This is because, on one hand, the algorithm runs a maximum of 4 times, the maximum number of rounds, but on the other hand, forcing only one location on a leg doesn't reduce the amount of trips the algorithm needs to consider every iteration. However, we can see that when 3 locations are forced, the average response time is 0.46 seconds faster than the response time without any forced conditions. The same thing can be seen when forcing 2 IMOs and one location, where the average response time is 0.2 seconds faster than the average response time without any conditions. This means that forcing a lot of conditions can lead to better performances. This is because the amount of possibilities to consider for every round are reduced a lot. This allows the journeys to be computed faster on average when forcing a lot of conditions.

Finally, the average response time when forcing a condition is on average 0.46 seconds longer than the average response time when not forcing conditions. This means that on average, forcing a condition increases the response time by 35 percents. The worst condition to force is only one location, which increases the average response time of 95 percents. Improvements could be made to the performances when forcing only one or two conditions. However, this would require Searoutes to modify the whole API to use an other algorithm, or to use a different way to force conditions on the returned journeys. The average response time is still reasonable, the API is usable, no matter what or how many conditions are forced. In addition to that, Searoutes specified that they didn't care about the performances to much, if the algorithm was reliable.

The only way to reduce the average time without modifying the way the whole API works would be to change the heuristic used to stop the loop that runs the algorithm over every Via composite. This would reduce the accuracy of the journeys given by the API, and this consequence doesn't fit Searoutes requirements.

Conclusion

Through this internship, I manage to modify one of Searoutes APIs. I had to study the base algorithm, called RAPTOR algorithm, and understand how Searoutes modified it to fit better in their API. Then, I had to find a way to implement the possibility to force specific conditions on the journeys computed by the API. Thereafter, I had to choose the best pattern to use, and properly implement every new functionality of the API. I worked on both the algorithmic aspect and technical aspect. Finally, I had to test these new functionality, by checking their behavior and their performances. During the internship, I learned how to use important developing and programming tools such as GitHub or a Java IDE. I had to understand the logic behind the Java language and how to properly develop in Java. Working in a company also allowed me to better understand how people can work together on a complex programming structure and to improve my knowledge in computer science.

The goal of the internship was achieved, every functionality is working and has been tested. The final product meets the requirements that Searoutes expected. However, with a longer period of time, a better approach to implement the conditions could have been found, which could have allowed a reduction of the impact of the forced conditions on the performances of the algorithm. Some queries can take up to double the base response time. This is only for a low amount of conditions. Improving this would probably lead to a different algorithm to be used, which means the whole structure of the API would need to be reworked. A way of improving the performances of the algorithm could be avoiding running it multiples times to force the conditions on different legs. This could be made using a different routing algorithm, as the RAPTOR way of exploring the network isn't optimal when it comes to forcing conditions. Once again, the goal of the internship was to improve the already existing implementation, and to use the RAPTOR algorithm as a base. The performances are fine for the need of the company, and the use the clients will make of this API. The forced conditions are only used to enhance the precision of the results.

This internship was a wonderful opportunity. I allowed me to be a part of Searoutes Tech team for a month. This internship fits my study project for the following years. The knowledge I gained during this internship will be useful in MPCI, and during the following years. I express my sincere appreciation to the entire Searoutes team for providing me this wonderful opportunity.

Bibliography

Références

[1] Delling D., Pajor T., Werneck R., "Round-Based Public Transit Routing", 2012 : https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/raptor_alenex.pdf

[2] Tutorial Java : https://www.baeldung.com/get-started-with-java-series

[3] Explanation of the composite structure : https://refactoring.guru/design-patterns/composite

Appendix

Algorithm 4: ForwardRaptorAlgorithmWithViaIMO

Input : P_s the starting point, P_t the target point, τ the date of the start, , $forcedImo$ the forced IMO

Parameters: K the number of rounds, T a set of trips, $ListOfVia$ generated thanks to $forcedImo$(mettre un lien à la partie où on l'explique)

Output : A set of Pareto-optimal journeys

```
for via ∈ ListOfVia do
    I ← round stored in via
    for k ← 1 to K do
        clear(Q) ;
        ; // collect trips reaching marked stops according to the IMO constraint
        for p inside marked_stops do
            for t inside T serving P do
                if (Imo(t) = forcedImo and k = I) OR (Imo(t) ≠ forcedImo and k ≠ I) then
                    if p' ∈ marked_stops st (t, p') exists and p' comes before p inside t then
                        add (t, p') to Q ;
                    else
                        add (t, p) to Q;
                    end
                end
            end
            remove p from marked_stops
        end
        ; // check if τ(p) can be improved
        for (t, p) ∈ Q do
            for p_i ∈ t after p do
                ; // update τ_k and τ*(p_i) if it improves globally
                if arr(t, p_i) < τ*(p_i) and arr(t, p_i) < τ*(p_t) then
                    τ_k(p_i) ← arr(t, p_i);
                    τ*(p_i) ← arr(t, p_i);
                    mark (p_i);
                end
                ; // update τ_k and τ*(p_i) if it improves only this round but using the right IMO
                 on the right round
                if k = I and arr(t, p_i) < τ_k(p_i) then
                    τ_k(p_i) ← arr(t, p_i);
                    τ*(p_i) ← arr(t, p_i);
                    mark (p_i);
                end
            end
        end
        if marked_stops is empty then
            break
        end
    end
end
Return ReconstructJourney()
```

Algorithm 5: ForwardRaptorAlgorithm with Via location

Input : P_s the starting point, P_t the target point, τ the date of the start
Parameters: K the number of rounds, T a set of trips
Output : A set of Pareto-optimal journeys

```
1  for k ← 1 to K do
2      clear(Q) ;
       ; // collect trips reaching marked stops
3      for p inside marked_stops do
4          for t inside T serving P do
5              if p' ∈ marked_stops st (t, p') exists and p comes before p' inside t then
6                  replace (t, p') by (t, p) in Q ;
7              else
8                  add (t, p) to Q;
9              end
10         end
11         remove p from marked_stops
12     end
       ; // check if τ(p) can be improved
13     for (t, p) ∈ Q do
14         for p_i ∈ t after p do
               ; // considers a stop if it matches the forced locations condition
15             if (p_i is a forced location and k is the round where p_i is forced) or (p_i is not a forced
                location and k isn't a round with a forced location) then
16                 if arr(t, p_i) < τ*(p_i) and arr(t, p_i) < τ*(p_t) then
17                     τ_k(p_i) ← arr(t, p_i) ;
18                     τ*(p_i) ← arr(t, p_i) ;
19                     mark (p_i) ;
20                 end
21             end
22         end
23     end
24     if marked_stops is empty then
25         break
26     end
27 end
28 Return ReconstructJourney()
```

Listing 7 – Java code for the Via interface

```java
public interface ViaInterface {

    boolean isCompatibleAccumulateStep(int currentRound, Trip trip);

    boolean isCompatibleTraverseStep(int currentRound, StopTime stopTime);

    boolean hasViaAtRound(int roundIndex);

    int minNRoundsToRespectVia();

    boolean hasVia();

    ViaInterface mirrorVia(int maxNbRounds);
}
```

Listing 8 – Java code for the Via composite class

```java
public class ViaComposite implements ViaInterface {

    List<ViaInterface> components = new ArrayList<>();

    public ViaComposite(List<ViaInterface> componentsToAdd) {
        this.components.addAll(componentsToAdd);
    }

    @Override
    public boolean isCompatibleAccumulateStep(int currentRound, Trip trip) {
        if (components == null || components.isEmpty()) {
            return true;
        }
        return components.stream().allMatch(viaInterface -> viaInterface.
            isCompatibleAccumulateStep(currentRound, trip));
    }

    @Override
    public boolean isCompatibleTraverseStep(int currentRound, StopTime stopTime) {
        if (components == null || components.isEmpty()) {
            return true;
        }
        return components.stream().allMatch(viaInterface -> viaInterface.
            isCompatibleTraverseStep(currentRound, stopTime));
    }

    @Override
    public boolean hasViaAtRound(int roundIndex) {
        if (components == null || components.isEmpty()) {
            return false;
        }
        return components.stream().anyMatch(viaInterface -> viaInterface.hasViaAtRound(
            roundIndex));
    }

    @Override
    public int minNRoundsToRespectVia() {
        if (components == null || components.isEmpty()) {
            return -1;
        }
```

```
        return components.stream().map(ViaInterface::minNRoundsToRespectVia).max(
            Integer::compareTo).orElse(-1);
    }

    @Override
    public boolean hasVia() {
        if (components == null || components.isEmpty()) {
            return false;
        }
        return components.stream().anyMatch(ViaInterface::hasVia);
    }

    @Override
    public ViaInterface mirrorVia(int maxNbRounds) {
        if (components == null || components.isEmpty()) {
            return new ViaComposite(new ArrayList<>());
        }
        List<ViaInterface> mirroredVias = new ArrayList<>();
        components.forEach(viaInterface -> {
            ViaInterface mirroredVia = viaInterface.mirrorVia(maxNbRounds);
            mirroredVias.add(mirroredVia);
        });
        return new ViaComposite(mirroredVias);
    }
}
```

Listing 9 – Java code for the Via IMO leaf class

```
public class ViaImoLeaf implements ViaInterface{

    Map<Integer, Integer> imoPerLeg = new HashMap<>();
    Map<Integer, Set<Integer>> blockedImosPerLeg = new HashMap<>();

    public ViaImoLeaf(Map<Integer, Integer> imoPerLeg, int maxRounds) {
        this.imoPerLeg = new HashMap<>(imoPerLeg);
        for (int i = 1; i <= maxRounds; i++) {
            this.blockedImosPerLeg.put(i, new HashSet<>());
        }
        for (Map.Entry<Integer, Integer> legNumberAndImo : imoPerLeg.entrySet()) {
            for (int i = 1; i <= maxRounds; i++) {
                if (i != legNumberAndImo.getKey()) {
                    this.blockedImosPerLeg.get(i).add(legNumberAndImo.getValue());
                }
            }
        }
    }

    @Override
    public boolean isCompatibleAccumulateStep(int currentRound, Trip trip) {
        int imo = trip.getImo();
        Integer imoOfLeg = this.imoPerLeg.get(currentRound);
        Set<Integer> blockedImosOfLeg = this.blockedImosPerLeg.get(currentRound);
        if (blockedImosOfLeg == null && imoOfLeg == null) {
            return true;
        }
        return (imoOfLeg != null && imo == imoOfLeg) || (imoOfLeg == null && !
            blockedImosOfLeg.contains(imo));
    }

```

```
    @Override
    public boolean isCompatibleTraverseStep(int currentRound, StopTime stopTime) {
        return true;
    }

    @Override
    public boolean hasViaAtRound(int roundIndex) {
        return imoPerLeg.containsKey(roundIndex);
    }

    @Override
    public int minNRoundsToRespectVia() {
        int minOfRounds = -1;
        for (Map.Entry<Integer, Integer> legNumberAndImo : imoPerLeg.entrySet()) {
            minOfRounds = Math.max(minOfRounds, legNumberAndImo.getKey());
        }
        return minOfRounds;
    }

    @Override
    public boolean hasVia() {
        return !imoPerLeg.isEmpty();
    }

    @Override
    public ViaInterface mirrorVia(int maxNbRounds) {
        // Create the "mirror" via, i.e. the new via that would give the same
            constraint in backward
        // (resp.forward) than the via in forward (resp.backward)
        Map<Integer, Integer> mirrorImoPerLeg = new HashMap<>();
        for (int i = 1; i <= maxNbRounds; i++) {
            Integer imoOfLeg = this.imoPerLeg.get(i);
            if (imoOfLeg != null) {
                mirrorImoPerLeg.put(maxNbRounds - i + 1, imoOfLeg);
            }
        }
        return new ViaImoLeaf(mirrorImoPerLeg, maxNbRounds);
    }
}
```

Listing 10 – Java code for the Via location leaf class

```
public class ViaLocationLeaf implements ViaInterface{

    Map<Integer, Locode> locationPerLeg = new HashMap<>();
    Map<Integer, Set<Locode>> blockedLocationPerLeg = new HashMap<>();

    public ViaLocationLeaf (Map<Integer, Locode> locationPerLeg, int maxRounds) {
        this.locationPerLeg = new HashMap<>(locationPerLeg);
        for (int i = 1; i <= maxRounds; i++) {
            this.blockedLocationPerLeg.put(i, new HashSet<>());
        }
        for (Map.Entry<Integer, Locode> legNumberAndLocation : locationPerLeg.entrySet
            ()) {
            for (int i = 1; i <= maxRounds; i++) {
                if (i != legNumberAndLocation.getKey()) {
                    this.blockedLocationPerLeg.get(i).add(legNumberAndLocation.getValue
                        ());
                }
```

```
            }
        }
    }

    @Override
    public boolean isCompatibleAccumulateStep(int currentRound, Trip trip) {
        return true;
    }

    @Override
    public boolean isCompatibleTraverseStep(int currentRound, StopTime stopTime) {
        Locode locode = stopTime.getStop().getLocode();
        Locode locationOfLeg = this.locationPerLeg.get(currentRound);
        Set<Locode> blockedLocationOfLeg = this.blockedLocationPerLeg.get(currentRound)
            ;
        if (blockedLocationOfLeg == null && locationOfLeg == null) {
            return true;
        }
        return (locationOfLeg != null && locode.equals(locationOfLeg)) || (
            locationOfLeg == null && !blockedLocationOfLeg.contains(locode));
    }

    @Override
    public boolean hasViaAtRound(int roundIndex) {
        return locationPerLeg.containsKey(roundIndex);
    }

    @Override
    public int minNRoundsToRespectVia() {
        int minOfRounds = -1;
        for (Map.Entry<Integer, Locode> legNumberAndLocation : locationPerLeg.entrySet
            ()) {
            minOfRounds = Math.max(minOfRounds, legNumberAndLocation.getKey());
        }
        return minOfRounds;
    }

    @Override
    public boolean hasVia() {
        return !locationPerLeg.isEmpty();
    }

    @Override
    public ViaInterface mirrorVia(int maxNbRounds) {
        // Create the "mirror" via, i.e. the new via that would give the same
            constraint in backward
        // (resp.forward) than the via in forward (resp.backward)
        Map<Integer, Locode> mirrorLocationPerLeg = new HashMap<>();
        for (int i = 1; i <= maxNbRounds; i++) {
            Locode locationOfLeg = this.locationPerLeg.get(i);
            if (locationOfLeg != null) {
                mirrorLocationPerLeg.put(maxNbRounds - i, locationOfLeg);
            }
        }
        return new ViaLocationLeaf(mirrorLocationPerLeg, maxNbRounds);
    }
}
```

Listing 11 – Java code for generateAllArrangement method

```
317     public static <T, U> List<Map<T, U>> generateAllArrangements(List<T> keys, List<U>
            values,
318             Integer currentIndex) {
319         //Recursive method that generates every arrangement possible of map with keys
                and values (Order and no repetition)
320         List<Map<T, U>> updatedList = new ArrayList<>();
321         if (currentIndex != values.size() - 1) {
322             List<Map<T, U>> currentList = generateAllArrangements(keys, values,
                    currentIndex + 1);
323             for (Map<T, U> map : currentList) {
324                 for (T key : keys) {
325                     if (map.get(key) == null) {
326                         Map<T, U> temporaryMap = new HashMap<>(map);
327                         temporaryMap.put(key, values.get(currentIndex));
328                         updatedList.add(temporaryMap);
329                     }
330                 }
331             }
332         } else {
333             for (T key : keys) {
334                 Map<T, U> map = new HashMap<>();
335                 map.put(key, values.get(currentIndex));
336                 updatedList.add(map);
337             }
338         }
339         return updatedList;
340     }
```

Listing 12 – Java code for a test of the Forward Raptor Algorithm

```
341     @Test
342     public void testForwardRaptorAlgorithmForcedImoAndLocation (){
343         //force "C" on first leg and imo 5 on second, takes 5 instead of 4 on second
                leg.
344         ForwardRaptorAlgorithm raptorAlgorithm = new ForwardRaptorAlgorithm(timetable);
345         RaptorSearchParameters raptorSearchParameters = new RaptorSearchParameters();
346         raptorSearchParameters.setSearchSource(stopsByName.get("A"));
347         raptorSearchParameters.setSearchDestination(stopsByName.get("D"));
348         raptorSearchParameters.setStartSearchDate(GeneralUtils.fromDateString("
                2023-08-01 10:00").getEpochSecond());
349         raptorSearchParameters.setEndSearchDate(Instant.MAX.getEpochSecond());
350         raptorSearchParameters.setNMaxRounds(2);
351         Map<Integer, Locode> viaMapLocode = new HashMap<>();
352         Map<Integer, Integer> viaMapImo = new HashMap<>();
353         viaMapImo.put(2, 5); //forcing imo 5 on first leg
354         viaMapLocode.put(1, stopsByName.get("C").getLocode()); //forcing location "C"
                on first leg
355         List<ViaInterface> components = new ArrayList<>();
356         components.add(new ViaImoLeaf(viaMapImo, 2));
357         components.add(new ViaLocationLeaf(viaMapLocode, 2));
358         raptorSearchParameters.setVia(new ViaComposite(components));
359         raptorSearchParameters.setMinTsTime(0);
360
361         List<Journey> journeys = raptorAlgorithm.run(raptorSearchParameters).getResults
                ();
362         assertEquals(1, journeys.size());
363         assertEquals(1, journeys.get(0).getLegs().get(0).getImo());
364         assertEquals(5, journeys.get(0).getLegs().get(1).getImo());
```

```
        assertEquals(stopsByName.get("A"), journeys.get(0).getLegs().get(0).
            getFromStopTime().getStop());
        assertEquals(stopsByName.get("C"), journeys.get(0).getLegs().get(0).
            getToStopTime().getStop());
        assertEquals(stopsByName.get("D"), journeys.get(0).getLegs().get(1).
            getToStopTime().getStop());
    }
}
```

Listing 13 – Java code for the test methods of the generateAllArrangement method written during the internship

```
public class GeneralUtilsTest {

    @Test
    public void testCreatingAllMapAvailableOneImoThreeRounds() {
        List<Integer> forcedImosList = new ArrayList<>();
        List<Integer> allRound = new ArrayList<>();
        forcedImosList.add(1);
        Set<Map<Integer, Integer>> expectedFromMethod = new HashSet<>();
        for (int i = 1; i < 4; i++) {
            allRound.add(i);
            Map<Integer, Integer> map = new HashMap<>();
            map.put(i, 1);
            expectedFromMethod.add(map);
        }
        List<Map<Integer, Integer>> MethodResult = GeneralUtils.generateAllArrangements
            (
                allRound, forcedImosList, 0);
        assertEquals(expectedFromMethod, new HashSet<>(MethodResult));
    }

    @Test
    public void testCreatingAllMapAvailableTwoImosTwoRounds() {
        List<Integer> forcedImosList = new ArrayList<>();
        List<Integer> allRound = new ArrayList<>();
        forcedImosList.add(1);
        forcedImosList.add(2);
        allRound.add(1);
        allRound.add(2);
        Set<Map<Integer, Integer>> expectedFromMethod = new HashSet<>();
        Map<Integer, Integer> map1 = new HashMap<>();
        Map<Integer, Integer> map2 = new HashMap<>();
        map1.put(1, 1);
        map1.put(2, 2);
        map2.put(1, 2);
        map2.put(2, 1);
        expectedFromMethod.add(map1);
        expectedFromMethod.add(map2);
        List<Map<Integer, Integer>> MethodResult = GeneralUtils.generateAllArrangements
            (
                allRound, forcedImosList, 0);
        assertEquals(expectedFromMethod, new HashSet<>(MethodResult));
    }

    @Test
    public void testCreatingAllMapAvailableTwoImoThreeRound() {
        List<Integer> forcedImosList = new ArrayList<>();
        List<Integer> allRound = new ArrayList<>();
```

```
        Set<Map<Integer, Integer>> expectedFromMethod = new HashSet<>();
        forcedImosList.add(1);
        forcedImosList.add(2);
        allRound.add(1);
        allRound.add(2);
        allRound.add(3);
        Map<Integer, Integer> map1 = new HashMap<>();
        Map<Integer, Integer> map2 = new HashMap<>();
        Map<Integer, Integer> map3 = new HashMap<>();
        Map<Integer, Integer> map4 = new HashMap<>();
        Map<Integer, Integer> map5 = new HashMap<>();
        Map<Integer, Integer> map6 = new HashMap<>();
        map1.put(1, 1);
        map1.put(2, 2);
        map2.put(1, 1);
        map2.put(3, 2);
        map3.put(2, 1);
        map3.put(1, 2);
        map4.put(2, 1);
        map4.put(3, 2);
        map5.put(3, 1);
        map5.put(1, 2);
        map6.put(3, 1);
        map6.put(2, 2);
        expectedFromMethod.add(map1);
        expectedFromMethod.add(map2);
        expectedFromMethod.add(map3);
        expectedFromMethod.add(map4);
        expectedFromMethod.add(map5);
        expectedFromMethod.add(map6);
        List<Map<Integer, Integer>> MethodResult = GeneralUtils.generateAllArrangements
            (
                allRound, forcedImosList, 0);
        assertEquals(expectedFromMethod, new HashSet<>(MethodResult));

    }
}
```

Listing 14 – Java code for implementation test methods written during the internship

```
    @Test
    public void two_imos_journeys_test() {
        String json =
                given().
                        param("fromLocode", "CNSHA").
                        param("toLocode", "NOOSL").
                        param("viaImos", "9762338,9354337").
                        param("searchMode", "FORWARD").
                        param("fromDate", "2023-08-01").
                        when().
                        get("/itinerary/v2/route").
                        then().
                        statusCode(200).
                        extract().asString();

        ReadContext jsonPath = JsonPath.parse(json);
        Integer nrJourneys = jsonPath.read("length()");
        for (int journeyIndex = 0; journeyIndex < nrJourneys;
                journeyIndex++) {
```

```
470             List<Integer> imosInJourney = jsonPath.read("$[" + journeyIndex + "].
                    features[*].properties.vessel.imo");
471             assertThat(imosInJourney, hasItem(9762338));
472             assertThat(imosInJourney, hasItem(9354337));
473 
474         }
475     }
476 
477     @Test
478     public void multiple_wrong_imo_test() {
479         String json =
480                 given().
481                         param("fromLocode", "CNSHA").
482                         param("toLocode", "NOOSL").
483                         param("viaImos", "1111,2222").
484                         param("searchMode", "FORWARD").
485                         param("fromDate", "2023-08-01").
486                         when().
487                         get("/itinerary/v2/route").
488                         then().
489                         statusCode(404).
490                         extract().asString();
491         ReadContext jsonPath = JsonPath.parse(json);
492 
493         assertThat(jsonPath.read("$.errorCodes").toString(), equalTo("[\"1031\"]"));
494         assertThat(jsonPath.read("$.messages").toString(), equalTo("[\"Vessel with imo
                '1111, 2222' not found.\"]"));
495     }
496 
497     @Test
498     public void combined_with_multiple_imo_test() {
499         String json =
500                 given().
501                         param("fromLocode", "CNSHA").
502                         param("toLocode", "NOOSL").
503                         param("viaImos", "9778806,9116187").
504                         param("searchMode", "COMBINED").
505                         param("fromDate", "2023-08-01").
506                         when().
507                         get("/itinerary/v2/route").
508                         then().
509                         statusCode(200).
510                         extract().asString();
511 
512         ReadContext jsonPath = JsonPath.parse(json);
513         Integer nrJourneys = jsonPath.read("length()");
514         for (int journeyIndex = 0; journeyIndex < nrJourneys;
515                 journeyIndex++) {
516             List<Integer> imosInJourney = jsonPath.read("$[" + journeyIndex + "].
                    features[*].properties.vessel.imo");
517             assertThat(imosInJourney, hasItem(9778806));
518             assertThat(imosInJourney, hasItem(9116187));
519         }
520     }
521 
522     @Test
523     public void combined_route_test_with_via_location(){
524         String json =
525                 given().
```

```
                        param("fromLocode", "CNSHA").
                        param("toLocode", "NOOSL").
                        param("viaLocations", "PTSIE").
                        param("searchMode", "COMBINED").
                        param("fromDate", "2023-08-01").
                        when().
                        get("/itinerary/v2/route").
                        then().
                        statusCode(200).
                        extract().asString();

        ReadContext jsonPath = JsonPath.parse(json);
        Integer nrJourneys = jsonPath.read("length()");
        for (int journeyIndex = 0; journeyIndex < nrJourneys;
                journeyIndex++) {
            String locationsInJourney = jsonPath.read("$[" + journeyIndex + "].features
                [0].properties.to.locode");
            assertThat("PTSIE", equalTo(locationsInJourney));
        }
    }

    @Test
    public void combined_route_test_with_multiple_via_location(){
        String json =
                given().
                        param("fromLocode", "CNSHA").
                        param("toLocode", "BEANR").
                        param("viaLocations", "SGSIN, ITGOA").
                        param("searchMode", "COMBINED").
                        param("fromDate", "2023-08-01").
                        when().
                        get("/itinerary/v2/route").
                        then().
                        statusCode(200).
                        extract().asString();

        ReadContext jsonPath = JsonPath.parse(json);
        Integer nrJourneys = jsonPath.read("length()");
        for (int journeyIndex = 0; journeyIndex < nrJourneys;
                journeyIndex++) {
            List<String> locationsInJourney = jsonPath.read("$[" + journeyIndex + "].
                features[*].properties.to.locode");
            assertThat(locationsInJourney, hasItem("SGSIN"));
            assertThat(locationsInJourney, hasItem("ITGOA"));
        }
    }

    @Test
    public void combined_route_test_with_via_location_and_via_imo(){
        String json =
                given().
                        param("fromLocode", "CNSHA").
                        param("toLocode", "NOOSL").
                        param("viaLocations", "MYTPP").
                        param("searchMode", "COMBINED").
                        param("viaImos", "9302243").
                        param("fromDate", "2023-07-01").
                        when().
                        get("/itinerary/v2/route").
```

```
                        then().
                        statusCode(200).
                        extract().asString();

        ReadContext jsonPath = JsonPath.parse(json);
        Integer nrJourneys = jsonPath.read("length()");
        for (int journeyIndex = 0; journeyIndex < nrJourneys;
                journeyIndex++) {
            List<String> locationsInJourney = jsonPath.read("$[" + journeyIndex + "].
                features[*].properties.to.locode");
            List<Integer> imosInJourneys = jsonPath.read("$[" + journeyIndex + "].
                features[*].properties.vessel.imo");
            assertThat(locationsInJourney, hasItem("MYTPP"));
            assertThat(imosInJourneys, hasItem(9302243));
        }
    }
}
```

Listing 15 – Java code for tests of the Backward Forward Raptor Algorithm written during the internship

```
    @Test
    public void testBackwardRaptorWithMultipleForcedImo() {
        BackwardRaptorAlgorithm raptorAlgorithm = new BackwardRaptorAlgorithm(timetable
            );

        //Normal Backwards should choose Trip5 first leg then trip4 second leg, but
            forcing imo2 of first leg and imo3 on second should modify the result
        RaptorSearchParameters raptorSearchParameters = new RaptorSearchParameters();
        raptorSearchParameters.setSearchSource(stopsByName.get("D"));
        raptorSearchParameters.setSearchDestination(stopsByName.get("A"));
        raptorSearchParameters.setStartSearchDate(GeneralUtils.fromDateString("
            2023-08-05 10:00").getEpochSecond());
        raptorSearchParameters.setEndSearchDate(Instant.MIN.getEpochSecond());
        raptorSearchParameters.setNMaxRounds(4);
        Map<Integer, Integer> viaMap = new HashMap<>();
        viaMap.put(2, 2); // Force imo 2 on first leg
        viaMap.put(1, 4); // Force imo 4 on second leg
        List<ViaInterface> components = new ArrayList<>();
        components.add(new ViaImoLeaf(viaMap, 4));
        raptorSearchParameters.setVia(new ViaComposite(components));
        raptorSearchParameters.setMinTsTime(0);

        List<Journey> journeys = raptorAlgorithm.run(raptorSearchParameters).getResults
            ();
        assertEquals(1, journeys.size());
        assertEquals(2, journeys.get(0).getLegs().size());
        assertEquals(2, journeys.get(0).getLegs().get(0).getImo());
        assertEquals(4, journeys.get(0).getLegs().get(1).getImo());


    }

    @Test
    public void testBackwardRaptorWithViaLocation() {
        BackwardRaptorAlgorithm raptorAlgorithm = new BackwardRaptorAlgorithm(timetable
            );
        //A to D, normal backward would answer trip 2 but forcing stop C would return
            trip 1 then trip 7

```

```
631         RaptorSearchParameters raptorSearchParameters = new RaptorSearchParameters();
632         raptorSearchParameters.setSearchSource(stopsByName.get("D"));
633         raptorSearchParameters.setSearchDestination(stopsByName.get("A"));
634         raptorSearchParameters.setStartSearchDate(GeneralUtils.fromDateString("
                2023-08-05 10:00").getEpochSecond());
635         raptorSearchParameters.setEndSearchDate(Instant.MIN.getEpochSecond());
636         raptorSearchParameters.setNMaxRounds(2);
637         Map<Integer, Locode> viaMap = new HashMap<>();
638         viaMap.put(1, stopsByName.get("C").getLocode()); //forcing location "C" on
                second leg
639         List<ViaInterface> components = new ArrayList<>();
640         components.add(new ViaLocationLeaf(viaMap, 2));
641         raptorSearchParameters.setVia(new ViaComposite(components));
642         raptorSearchParameters.setMinTsTime(0);
643
644         List<Journey> journeys = raptorAlgorithm.run(raptorSearchParameters).getResults
                ();
645         assertEquals(1, journeys.size());
646         assertEquals(2, journeys.get(0).getLegs().size());
647         assertEquals(1, journeys.get(0).getLegs().get(0).getImo());
648         assertEquals(7, journeys.get(0).getLegs().get(1).getImo());
649
650         assertEquals(stopsByName.get("A"), journeys.get(0).getLegs().get(0).
                getFromStopTime().getStop());
651         assertEquals(stopsByName.get("C"), journeys.get(0).getLegs().get(0).
                getToStopTime().getStop());
652         assertEquals(stopsByName.get("C"), journeys.get(0).getLegs().get(1).
                getFromStopTime().getStop());
653         assertEquals(stopsByName.get("D"), journeys.get(0).getLegs().get(1).
                getToStopTime().getStop());
654     }
655
656     @Test
657     public void testBackwardRaptorWithViaLocationAndImoNoSolution() {
658         BackwardRaptorAlgorithm raptorAlgorithm = new BackwardRaptorAlgorithm(timetable
                );
659         //A to D, normal backward would answer trip 2 but forcing stop C and imo 2
                would not work, even if forcing stop C
660         //or imo 2 separately would give answer.
661
662         RaptorSearchParameters raptorSearchParameters = new RaptorSearchParameters();
663         raptorSearchParameters.setSearchSource(stopsByName.get("D"));
664         raptorSearchParameters.setSearchDestination(stopsByName.get("A"));
665         raptorSearchParameters.setStartSearchDate(GeneralUtils.fromDateString("
                2023-08-05 10:00").getEpochSecond());
666         raptorSearchParameters.setEndSearchDate(Instant.MIN.getEpochSecond());
667         raptorSearchParameters.setNMaxRounds(2);
668         Map<Integer, Locode> viaMapLocode = new HashMap<>();
669         Map<Integer, Integer> viaMapImo = new HashMap<>();
670         viaMapImo.put(1, 2); //forcing imo 2 on second leg
671         viaMapLocode.put(1, stopsByName.get("C").getLocode()); //forcing location "C"
                on second leg
672         List<ViaInterface> components = new ArrayList<>();
673         components.add(new ViaImoLeaf(viaMapImo, 2));
674         components.add(new ViaLocationLeaf(viaMapLocode, 2));
675         raptorSearchParameters.setVia(new ViaComposite(components));
676         raptorSearchParameters.setMinTsTime(0);
677
678         List<Journey> journeys = raptorAlgorithm.run(raptorSearchParameters).getResults
```

```
            ();
679         assertEquals(0, journeys.size());
680     }
681
682     @Test
683     public void testBackwardRaptorWithViaLocationAndImo() {
684         BackwardRaptorAlgorithm raptorAlgorithm = new BackwardRaptorAlgorithm(timetable
                );
685         //A to D, normal backward would answer trip 2 but forcing stop C and imo 7
                force trip 1 then 7
686
687         RaptorSearchParameters raptorSearchParameters = new RaptorSearchParameters();
688         raptorSearchParameters.setSearchSource(stopsByName.get("D"));
689         raptorSearchParameters.setSearchDestination(stopsByName.get("A"));
690         raptorSearchParameters.setStartSearchDate(GeneralUtils.fromDateString("
                2023-08-05 10:00").getEpochSecond());
691         raptorSearchParameters.setEndSearchDate(Instant.MIN.getEpochSecond());
692         raptorSearchParameters.setNMaxRounds(2);
693         Map<Integer, Locode> viaMapLocode = new HashMap<>();
694         Map<Integer, Integer> viaMapImo = new HashMap<>();
695         viaMapImo.put(1, 7); //forcing imo 7 on second leg
696         viaMapLocode.put(1, stopsByName.get("C").getLocode()); //forcing location "C"
                on first leg
697         List<ViaInterface> components = new ArrayList<>();
698         components.add(new ViaImoLeaf(viaMapImo, 2));
699         components.add(new ViaLocationLeaf(viaMapLocode, 2));
700         raptorSearchParameters.setVia(new ViaComposite(components));
701         raptorSearchParameters.setMinTsTime(0);
702
703         List<Journey> journeys = raptorAlgorithm.run(raptorSearchParameters).getResults
                ();
704         assertEquals(1, journeys.size());
705         assertEquals(2, journeys.get(0).getLegs().size());
706         assertEquals(1, journeys.get(0).getLegs().get(0).getImo());
707         assertEquals(7, journeys.get(0).getLegs().get(1).getImo());
708
709         assertEquals(stopsByName.get("A"), journeys.get(0).getLegs().get(0).
                getFromStopTime().getStop());
710         assertEquals(stopsByName.get("C"), journeys.get(0).getLegs().get(0).
                getToStopTime().getStop());
711         assertEquals(stopsByName.get("C"), journeys.get(0).getLegs().get(1).
                getFromStopTime().getStop());
712         assertEquals(stopsByName.get("D"), journeys.get(0).getLegs().get(1).
                getToStopTime().getStop());
713     }
714         @Test
715     public void testForwardRaptorAlgorithmMultipleImo() {
716         ForwardRaptorAlgorithm raptorAlgorithm = new ForwardRaptorAlgorithm(timetable);
717         Map<Integer, Integer> viaMap = new HashMap<>();
718         viaMap.put(1, 2); // Force imo 2 on first leg
719         viaMap.put(2, 4); // Force imo 4 on second leg
720
721         // We want to force imo 2 in round 1 and then force imo 4 in round 4
722         RaptorSearchParameters raptorSearchParameters = new RaptorSearchParameters();
723         raptorSearchParameters.setSearchSource(stopsByName.get("A"));
724         raptorSearchParameters.setSearchDestination(stopsByName.get("C"));
725         raptorSearchParameters.setStartSearchDate(GeneralUtils.fromDateString("
                2023-07-10 00:00").getEpochSecond());
726         List<ViaInterface> components = new ArrayList<>();
```

```
727             components.add(new ViaImoLeaf(viaMap, 3));
728             raptorSearchParameters.setVia(new ViaComposite(components));
729             raptorSearchParameters.setEndSearchDate(Instant.MAX.getEpochSecond());
730             raptorSearchParameters.setNMaxRounds(3);
731             raptorSearchParameters.setMinTsTime(0);
732 
733             //No solution here, because imo 2 don't start in A
734             List<Journey> journeys = raptorAlgorithm.run(raptorSearchParameters).getResults
                    ();
735             assertEquals(0, journeys.size());
736 
737             //Only one solution here, the B-D-C with trip 2 then 4
738             raptorSearchParameters.setSearchSource(stopsByName.get("B"));
739             journeys = raptorAlgorithm.run(raptorSearchParameters).getResults();
740             assertEquals(1, journeys.size());
741             assertEquals(2, journeys.get(0).getLegs().get(0).getImo());
742             assertEquals(4, journeys.get(0).getLegs().get(1).getImo());
743             assertEquals(stopsByName.get("B"), journeys.get(0).getLegs().get(0).
                    getFromStopTime().getStop());
744             assertEquals(stopsByName.get("D"), journeys.get(0).getLegs().get(0).
                    getToStopTime().getStop());
745             assertEquals(stopsByName.get("C"), journeys.get(0).getLegs().get(1).
                    getToStopTime().getStop());
746 
747             //We test if the best solution without via is avoided, without via is B-D with
                    trip 1
748             //with via, is B-C-D with trip 1 then 4
749             raptorSearchParameters.setSearchDestination(stopsByName.get("D"));
750             viaMap = new HashMap<>();
751             viaMap.put(1, 1); // Force imo 2 on first leg
752             viaMap.put(2, 4); // Force imo 4 on second leg
753             components = new ArrayList<>();
754             components.add(new ViaImoLeaf(viaMap, 3));
755             raptorSearchParameters.setVia(new ViaComposite(components));
756             journeys = raptorAlgorithm.run(raptorSearchParameters).getResults();
757             assertEquals(1, journeys.size());
758             assertEquals(1, journeys.get(0).getLegs().get(0).getImo());
759             assertEquals(4, journeys.get(0).getLegs().get(1).getImo());
760             assertEquals(stopsByName.get("B"), journeys.get(0).getLegs().get(0).
                    getFromStopTime().getStop());
761             assertEquals(stopsByName.get("C"), journeys.get(0).getLegs().get(0).
                    getToStopTime().getStop());
762             assertEquals(stopsByName.get("D"), journeys.get(0).getLegs().get(1).
                    getToStopTime().getStop());
763         }
764 
765         @Test
766         public void testForwardRaptorAlgorithmForcedPort() {
767             ForwardRaptorAlgorithm raptorAlgorithm = new ForwardRaptorAlgorithm(timetable);
768             RaptorSearchParameters raptorSearchParameters = new RaptorSearchParameters();
769             Map<Integer, Locode> viaMap = new HashMap<>();
770             viaMap.put(1, stopsByName.get("C").getLocode()); // Force port "A" on first leg
771             List<ViaInterface> components = new ArrayList<>();
772             components.add(new ViaLocationLeaf(viaMap, 2));
773             raptorSearchParameters.setVia(new ViaComposite(components));
774             raptorSearchParameters.setSearchSource(stopsByName.get("B"));
775             raptorSearchParameters.setSearchDestination(stopsByName.get("D"));
776             raptorSearchParameters.setStartSearchDate(GeneralUtils.fromDateString("
                    2023-07-10 00:00").getEpochSecond());
```

```
777         raptorSearchParameters.setEndSearchDate(Instant.MAX.getEpochSecond());
778         raptorSearchParameters.setNMaxRounds(2);
779         raptorSearchParameters.setMinTsTime(0);
780         List<Journey> journeys = raptorAlgorithm.run(raptorSearchParameters).getResults
                ();
781
782         //Choose trip 1 instead of 2 because forced to go to "C"
783
784         assertEquals(1, journeys.size());
785         assertEquals(1, journeys.get(0).getLegs().get(0).getImo());
786         assertEquals(1, journeys.get(0).getLegs().get(1).getImo());
787         assertEquals(stopsByName.get("B"), journeys.get(0).getLegs().get(0).
                getFromStopTime().getStop());
788         assertEquals(stopsByName.get("C"), journeys.get(0).getLegs().get(0).
                getToStopTime().getStop());
789         assertEquals(stopsByName.get("D"), journeys.get(0).getLegs().get(1).
                getToStopTime().getStop());
790     }
791
792     @Test
793     public void testForwardRaptorAlgorithmForcedPortNoSolution() {
794         ForwardRaptorAlgorithm raptorAlgorithm = new ForwardRaptorAlgorithm(timetable);
795         RaptorSearchParameters raptorSearchParameters = new RaptorSearchParameters();
796         Map<Integer, Locode> viaMap = new HashMap<>();
797         viaMap.put(1, stopsByName.get("A").getLocode()); // Force port "A" on first leg
798         List<ViaInterface> components = new ArrayList<>();
799         components.add(new ViaLocationLeaf(viaMap, 2));
800         raptorSearchParameters.setVia(new ViaComposite(components));
801         raptorSearchParameters.setSearchSource(stopsByName.get("B"));
802         raptorSearchParameters.setSearchDestination(stopsByName.get("D"));
803         raptorSearchParameters.setStartSearchDate(GeneralUtils.fromDateString("
                2023-07-10 00:00").getEpochSecond());
804         raptorSearchParameters.setEndSearchDate(Instant.MAX.getEpochSecond());
805         raptorSearchParameters.setNMaxRounds(2);
806         raptorSearchParameters.setMinTsTime(0);
807         List<Journey> journeys = raptorAlgorithm.run(raptorSearchParameters).getResults
                ();
808
809         //No answer because forced to go "A"
810
811         assertEquals(0, journeys.size());
812     }
813
814     @Test
815     public void testForwardRaptorAlgorithmForcedImoAndLocation (){
816         //force "C" on first leg and imo 5 on second, takes 5 instead of 4 on second
                leg.
817         ForwardRaptorAlgorithm raptorAlgorithm = new ForwardRaptorAlgorithm(timetable);
818         RaptorSearchParameters raptorSearchParameters = new RaptorSearchParameters();
819         raptorSearchParameters.setSearchSource(stopsByName.get("A"));
820         raptorSearchParameters.setSearchDestination(stopsByName.get("D"));
821         raptorSearchParameters.setStartSearchDate(GeneralUtils.fromDateString("
                2023-08-01 10:00").getEpochSecond());
822         raptorSearchParameters.setEndSearchDate(Instant.MAX.getEpochSecond());
823         raptorSearchParameters.setNMaxRounds(2);
824         Map<Integer, Locode> viaMapLocode = new HashMap<>();
825         Map<Integer, Integer> viaMapImo = new HashMap<>();
826         viaMapImo.put(2, 5); //forcing imo 5 on first leg
827         viaMapLocode.put(1, stopsByName.get("C").getLocode()); //forcing location "C"
```

```
            on first leg
828         List<ViaInterface> components = new ArrayList<>();
829         components.add(new ViaImoLeaf(viaMapImo, 2));
830         components.add(new ViaLocationLeaf(viaMapLocode, 2));
831         raptorSearchParameters.setVia(new ViaComposite(components));
832         raptorSearchParameters.setMinTsTime(0);
833
834         List<Journey> journeys = raptorAlgorithm.run(raptorSearchParameters).getResults
                ();
835         assertEquals(1, journeys.size());
836         assertEquals(1, journeys.get(0).getLegs().get(0).getImo());
837         assertEquals(5, journeys.get(0).getLegs().get(1).getImo());
838         assertEquals(stopsByName.get("A"), journeys.get(0).getLegs().get(0).
                getFromStopTime().getStop());
839         assertEquals(stopsByName.get("C"), journeys.get(0).getLegs().get(0).
                getToStopTime().getStop());
840         assertEquals(stopsByName.get("D"), journeys.get(0).getLegs().get(1).
                getToStopTime().getStop());
841     }
842 }
843 }
```

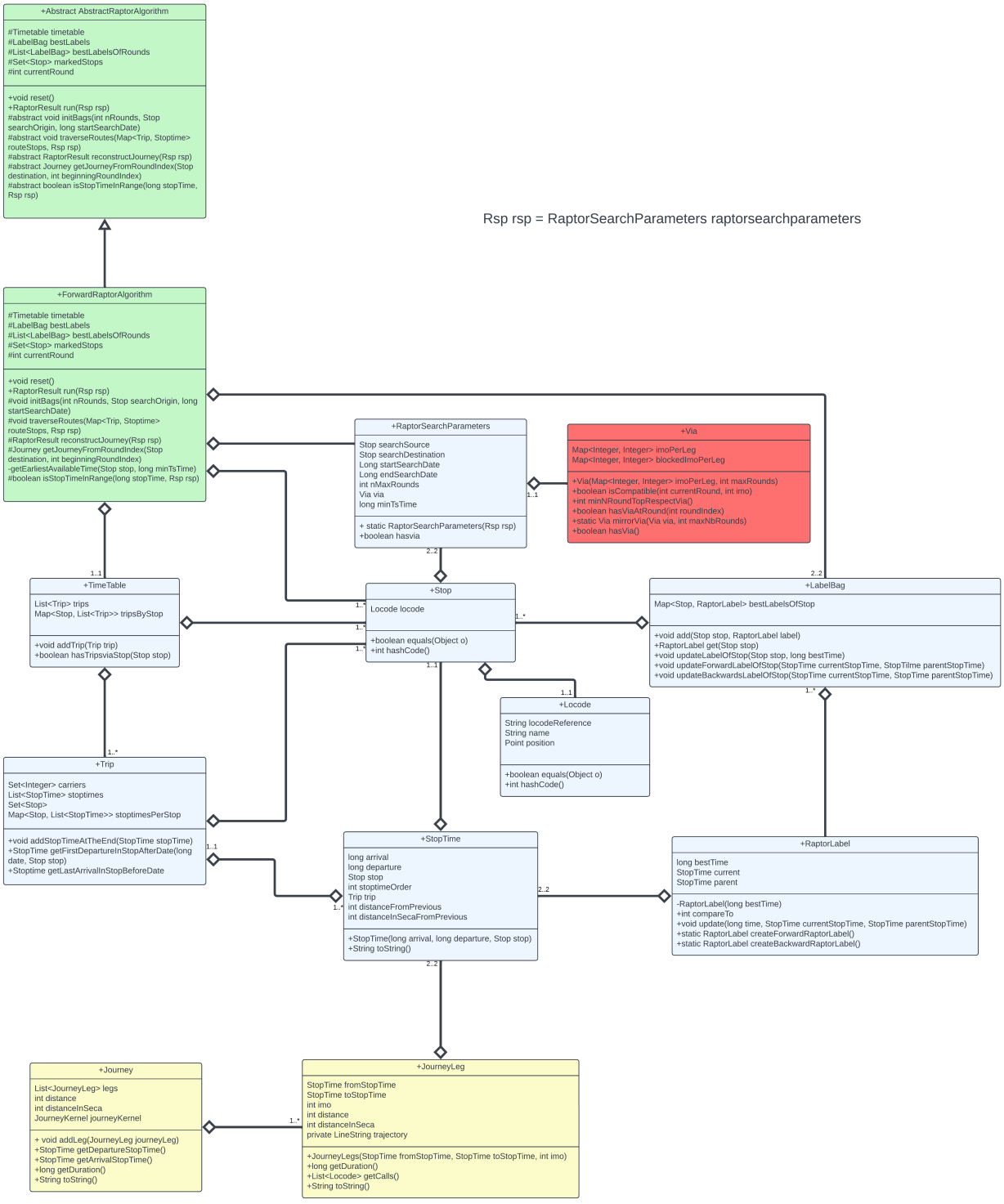

FIGURE 7 – UML diagram showing classes used in the app to run the algorithm properly