Defence Research and Development Canada
Recherche et développement pour la défense Canada

DRDC | RDDC
technologysciencetechnologie

PLANNING TOOL ALGORITHM IMPLEMENTATION FOR THE COURSE OF ACTION TEST BED

Laurence Olivier Marion-Ouellet
Éric-Olivier Bossé
Michel Mayrand
OODA Technologies

Prepared by:
OODA Technologies
4710 St-Ambroise, suite 226
Montreal, (Quebec)
Canada H4C 2C7

PSPC Contract Number: W7707-145677
Technical Authority: Tim Hammond, Defence Scientist
Contractor's date of publication: May 2018

Defence Research and Development Canada

Contract Report
DRDC-RDDC-2018-C183
September 2018

Canada

IMPORTANT INFORMATIVE STATEMENTS

This document was reviewed for Controlled Goods by Defence Research and Development Canada using the Schedule to the *Defence Production Act*.

RISOMIA: Call-up 20

PLANNING TOOL ALGORITHM IMPLEMENTATION FOR THE COURSE OF ACTION TEST BED

Prepared by

OODA Technologies
Laurence Olivier Marion-Ouellet
Éric-Olivier Bossé
Michel Mayrand

May 31st, 2018

Table of Content

1 Introduction

This call-up is in support of the ongoing Course of Action Test bed (COA-T), a system meant to test concepts for naval mission planning.

The output of this call-up is a set of three services:

1. The first is an auto-scheduling service for fuel optimization along a given route. This service is COA-T compatible. It receives requests containing the route, mission tasks (and their constraints), start and end times for the mission, and configuration parameters to output a schedule ensuring all the mission tasks can be completed and that the minimum quantity of fuel is used. If the mission tasks cannot be completed due to the weather or to constraints that cannot be met, the user is notified by an error message.

2. The second is a service that makes ShipMo7 predictions along a given route quickly accessible. It is also COA-T compatible. This service provides access to pre-computed predictions from ShipMo7, which have been archived in a PostgreSQL database. These predictions include the water resistance in irregular waves, as well as frequencies of keel emergence, deck wetness and Motion Induced Interruption. These parameters are accessed by specifying the ship type used (KINGSTON or HALIFAX class), the wave height, the wave frequency, the sea heading and finally the ship speed.

3. The last is a service that finds the shortest route between two points on a nautical chart, considering ocean depths and the ship's draught. It is COA-T compatible. A user sends a request to the system by specifying the start point, end point and two factors specifying the level of precision (*decimal* and *precision*, defined later in the appropriate section). The service will then answer with the length of the path in kilometres and a list of waypoints describing the itinerary.

Requesters for each of these services have also been created, in order to build the appropriate requests and verify responses from the services. Ideally, DRDC would integrate the creation of these requests through a GUI.

1.1 Document Purpose

The objective of this document is to provide a User/Administor/Developer Guide for the services developed during this call-up.

1.2 Document Overview

This document is intended for DRDC Atlantic scientists. It is organised as follows: Sections 2 to 5 present the three services developed, the design decisions and a quick user/administrator guide. Section 6 presents the lessons learned and the conclusions of this project which could prove useful for continuation of this call-up. Some annexes provide additional information on the necessary modifications to the data model currently used by DRDC for COA-T and also on how

to configure the different services through *application.properties* files or parameter options passed at startup.

1.3 Definitions, Acronyms and Abbreviations

COA-T	Course of Action Test bed
DBMS	Database Management System
DRDC	Defence Research and Development Canada
FOSS	Free and Open Source Software
IDE	Integrated Development Environment
PSA	Predictive Situational Awareness
RCN	Royal Canadian Navy
RPM	Revolutions per Minute
SOW	Statement of Work
TA	Technical Authority

2 Installation

2.1 Pre-requirements

Java 1.8 needs to be installed before installing the rest of the system. Also, the environment variable *JAVA_HOME* must be set properly and the *PATH* must contain the path to the java binaries (*java* and *javac*).

2.2 Maven

1. Check if Maven is not already installed by typing ***mvn –v*** at a command prompt. If the first line is not of type *Apache Maven X.X.X*, follow the rest of these steps.
2. Download Maven from https://maven.apache.org/download.cgi and follow the instructions listed in https://maven.apache.org/install.html
3. Unzip the package in a location of your choosing (ex*: C:\Program Files*)
4. Add the bin directory of the created directory *apache-maven-3.5.3* to the PATH environment variable
5. Confirm with ***mvn -v*** in a command prompt. The output should begin with *Apache Maven X.X.X*.

2.3 PostgreSQL Installation

1. Check if PostgreSQL is not already installed by typing ***psql --help*** at a command prompt. If installed, the output should be a list of possible commands. If not, follow the rest of these steps.
2. If you have access to Internet:
   - Go to http://www.postgresql.org/download/windows
   - Click on the link to "Download the installer," which will take you to: http://www.enterprisedb.com/products-services-training/pgdownload#windows.
   - Select the appropriate version of PostgreSQL and of your operating system in the download interface.
   - Click ***Download Now***.
3. If you do not have access to Internet and the installation executables (PostgreSQL and PostGIS) are provided via BISCOM , Sharepoint or a DVD:
   - Copy the provided zip file on the computer that will be used as the database server and unzip the zip file.
4. Navigate to where the installation file (named something like *postgresql-X.X.X-X-windows.exe*) is located and double-click on it to launch the installer.
5. Click ***Next*** through the first few Setup wizard steps. On the ***Password*** screen, type *postgres*. Note: PostgreSQL has its own users and superusers, which are separate and distinct from Windows users and administrators. By default, PostgreSQL installation sets

up both a PostgreSQL superuser named *postgres* and a Windows system user account named *postgres*. Both account names also use *postgres* as the password.

6. Click ***Next*** through the rest of the Setup wizard steps, leaving all values as their defaults (such as the Port).
7. The last step of the Setup wizard should ask if you want to launch something called ***Stack Builder***.
8. Make sure that the checkbox for ***Stack Builder*** is checked.
9. Click ***Finish***, which both finishes PostgreSQL installation and launches the Stack Builder.
10. On the first page of the ***Stack Builder*** window, in the drop-down menu, select ***PostgreSQL on port 5432***.
11. Click ***Next***
12. On the next page, click on the "+" sign next to ***Spatial Extensions***
13. Check the box next to "PostGIS 2.0 for PostgreSQL 9.X v2.X.X
14. Click ***Next*** until presented with the "PostGIS 2.X.X" setup window
15. If the PostGIS installation does not proceed because of firewall issues, then use the PostGIS executable by double clicking the application. The default parameters are usually sufficient for a successful installation but feel free to adjust some parameters if recommended by your network administrator. Please note that PostGIS requires a successful installation of PostgreSQL prior to its installation.

Some additional adjustments may be required within the DRDC local network, depending on the context (permission, network access, proxy, etc.), see your network administrator for further assistance.

2.4 Importing ShipMo7 data to PostgreSQL

Since the ShipMo7 and Scheduler services both need access to ShipMo7 data in order to gain access to water resistance, keel emergence, deck wetness and motion-induced interruption frequencies, the appropriate information must be loaded into a PostgreSQL database accessible to our services. The selection of PostgreSQL as the DBMS for the services was a requirement of the COA-T administrator.

In order to load the ShipMo7 database into your PostgreSQL server, first unzip the compressed database file (*shipmodat_postgresql_database_20180508.zip*) into the folder of your choice, then from this folder open a command prompt and create an empty database in your server with the command:

```
createdb -U postgres shipmodata
```

Then import the database file with the following command:

```
psql -U postgres -d shipmodata <  shipmodata.sql
```

If you use a PostgreSQL interface tool (e.g., PgAdmin III), just create a new database named *shipmodata* and import the SQL file shipmodata.sql into it.

Note: in pgAdmin 4 there is no "import" function. Instead, create the database and then use the Query Tool (accessed via a right click on the database) and from there load the SQL file and execute it.

2.5 COA-T Services Unpacking

2.5.1 Using the command prompt

1. Requirement: Pre-install Java 1.8, Maven 3 according to the instructions given above. Make sure that the JAVA environment variables and the path for accessing Java and Maven binaries are set correctly.
2. If you are running without internet, unzip the Maven (M2) deliverable in the Windows or Linux User home directory. It contains all the JAR files required for compiling the code offline.
3. Edit the following configuration files (see Annex C) in order to reflect your network parameters:
    a. *./COATscheduler/src/main/resources/application.properties*
    b. *./COATshipmo7 /src/main/resources/application.properties*
    c. *./COATShortestPath/src/main/resources/application.properties*

4. Access each of the following services (*COATscheduler, CoatShipMo7 and CoatShortestPath*) and run the command prompt command:
    ```
    mvn clean install
    ```

5. The executable JAR will be found in the target directory of the corresponding service.
6. Note that the Requester type services were designed to be run from within an Eclipse-like development environment, as they were created for testing purposes.

2.5.2 Using Eclipse

1. Requirement: Pre-install Java 1.8, Maven 3 (see above) and Eclipse IDE. Make sure that the JAVA environment variables and the path for accessing Java and Maven binaries are set correctly.
2. If you are running without internet, unzip the Maven (M2) deliverable in the Windows or Linux User home directory. It contains all the JAR files required for compiling the code offline.
3. Open Eclipse and import the services as a Maven Project. The use of Eclipse is the main way to use the Requester for testing each of the services. Each of the Requester modules must be imported in Eclipse and executed from there (after the corresponding service has been started).
4. Before executing the services, modify the following configuration files (see Annex C) in order to reflect your network parameters:
    a. *./COATscheduler/src/main/resources/application.properties*
    b. *./COATshipmo7/shipmo7/src/main/resources/application.properties*
    c. *./COATShortestPath/src/main/resources/application.properties*

2.6 GDAL

GDAL[1] is a set of libraries allowing access to the S57 maps. It is therefore necessary to the ShortestPath service. Instructions on how to install it were provided by DRDC and have been delivered along with this report under the name “GDAL Install Guide”.

[1] https://www.gdal.org/

3 AutoScheduler Service

3.1 Preliminary Requirements

The call-up SOW provided the initial set of requirements:

1. Implement into COA-T the *AutoScheduler* developed by DRDC.
2. Using FOSS, develop a fuel optimization algorithm that includes ShipMo7 data.
3. Communicate any recommendation deemed necessary to the ActiveMQ Data Model or the Database Model

3.2 Solution Overview

A COA-T service has been created which awaits *SchedulerRequest* objects. These objects contain a route (represented as a list of waypoints), start and end times for the mission, and a list of mission tasks. The mission tasks can have spatial and time constraints as well as weather constraints (e.g., the wind must be lower than 50 kph and the waves smaller than 2 meters). All mission tasks have a fixed duration. Other parameters included in the *SchedulerRequest* are the type of ship used (currently KINGSTON or HALIFAX class) and finally whether or not fuel optimization should be applied.

If fuel optimization is set to false, then the Scheduler will ingest the request and output a *ScheduleResponse* through ActiveMQ. This response tells the user if the operation was a success or not, if the fuel optimization was applied, notes (if they are needed) and, most importantly, a list of *ScheduleAction* objects. This time ordered list describes the steps the vessel needs to take in order to respect the mission tasks constraints. Included in this are the speed to apply, the task to perform (which may just be a “Move” action, where the ship is in transit), the heading, how long the task will last and how much distance will be covered during it. This list could then readily be used by a GUI to display a schedule for the vessel in question. If the weather prevents a valid schedule from being produced, then appropriate error messages are also sent.

If fuel optimization is activated, the Scheduler will also add the appropriate motor configurations to use in each of the move *ScheduleActions*. The motor configurations are selected to both respect the effective speed that will be needed to satisfy the schedule (weather effects can slow down the ship) and also to minimize the consumption of fuel. Selecting this function takes predictions from ShipMo7 into account. Should predictions from ShipMo7 suggest that some of the mission tasks would not be achievable in the available time because wave conditions are expected to slow the ship, appropriate error messages will be included in the response.

3.3 Design

In this section, we will overview the sequence of events leading to the creation of a *ScheduleResponse* from the reception of a request.

1. The Spring java app needs to be running, listening on the appropriate ActiveMQ topic. To verify that this service is started, look for the heartbeat messages that this app sends periodically to the *C2.STATUS* topic.

2. A message is sent by either the provided *ScheduleRequester* or by a GUI implementing the same methods, and is then received through the monitored topic by the *Consumer.class*. If the *JMSType* of this message is *SCHEDULE_REQUEST*, we parse it as a *SchedulerRequest* object and create a new *RequestProcessor* object, which will take care of the request.
3. The *RequestProcessor* pre-processes some parts of the *SchedulerRequest* (More details in section 3.3.1) and then gives it to a *ScheduleProcessor* object.
4. A *ScheduleProcessor* object is created and used to generate different combinations of move actions and tasks until success is achieved or until five different orders have been tried and failed. Success in this case is defined by the successful creation of a schedule in which the tasks can be performed according to their weather and time constraints. Section 3.3.2 describes this step.
5. If the previous step is successful, then a valid schedule is produced and *ScheduleActions* are built, describing when the tasks and "Move" actions of the ship will occur. These actions are then split into smaller parts, as necessary, so that all these parts have constant headings.
6. A *FuelOptimizer* object then takes care of splitting the "Move" parts of the schedule into sub parts where the motor configuration of the ship can be optimized to save fuel/give a better estimate of the actual fuel consumption.
7. The final step is to calculate the drift of the ship accounting for wind and current effects before sending the response in the form of a JSON-serialized *ScheduleResponse* object, the format of which is described in Annex A. These drift calculations are performed after the fuel optimization process rather than included within due to a lack of information surrounding the effects of wind and current on ShipMo7 calculations and how one affects another. Another concern raised was the effect of using a great circle trajectory, meaning a continuous change in a ship's heading. Following discussions in April, it was deemed better to inform a navigation officer of the continuous effects of the drift rather than try to apply corrections.

3.3.1 RequestProcessor

1. The *RequestProcessor* takes the different waypoints of the route as input to build the legs of the trip and to define the general area of the mission. The mission area is defined by a *LatLonBox* object.
2. Now that we have a *LatLonBox* delimiting the mission area, it is possible to request the weather for this zone. Weather requests (which include wind, waves and currents) are created and sent and the responses are awaited. In the case of a timeout or if only partial data is received, 2 default forecasts (in which all condition measurements are set to 0) are added for every type of weather. The goal of these additions is to prevent errors when interpolating the weather to a particular point.
3. Due to requirements of the DRDC *AutoScheduler* algorithm, which uses *SpatialInterval* and *TimeInterval* objects, the route of the ship needs to be described as a function of arclength, effectively converting it to a one-dimensional (1D) object. This 1D trajectory indicates the ship's progress along the original 2D route on the surface of the ocean. By the same token, constraints limiting where mission tasks can be accomplished, which are most naturally defined using 2D regions on the surface of the ocean, are converted into constraints on arclength. This is done for a given route by finding the arclength intervals

over which the route lies in the constraint areas. More specifically, the original route is broken up into a list of small *Step* objects, which correspond to points along the 1D trajectory. From these spatial points, the spatial availability of the mission tasks is translated from *LatLon* requirements into 1D intervals. Once both the route and the mission task spatial constraints are described with such 1D intervals, it becomes possible to use the *AutoScheduler*.

4. The information is then sent to a *ScheduleProcessor* object, which will attempt to build a schedule, represented in our app by a list of *ScheduleAction*s. In case of failure, the *RequestProcessor* takes care of sending the appropriate response.
5. These *ScheduleActions* are then split, in order to recreate the necessary waypoints. For example, where a *ScheduleAction* previously ordered to simply "Move" for a thousand miles, this one could be split in order to reflect its two sub-components: "Move" for two hundred miles at 36 degrees heading and then "Move" for eight hundred miles at a 74 degree heading, for example.
6. If fuel optimization is called for, the next step is to load the list of *ScheduleActions* and then send them to a *FuelOptimizer* object. This *FuelOptimizer* will further subdivide the 'Move' *ScheduleActions*, recommending a motor configuration to use in each one in order to save fuel. It will also provide the expected fuel consumption for the entire journey. The last information added to these *ScheduleActions* is the average drift that will be applied on the ship due to current and wind effects for each *ScheduleAction*.
7. The final step is to compile these *ScheduleAction*s in a message and send it to the appropriate topic in COA-T.

3.3.2 ScheduleProcessor

The responsibility of the *scheduleProcessor* is to take the mission tasks and any location and time constraints these may have and from these to build a list describing when and where the tasks could be done in order to respect these constraints. This is done as an iterative process because weather data are needed to know if a schedule is possible, but a schedule is needed to know the weather data. The following paragraphs describe the process in detail:

1. The first method of a *ScheduleProcessor* object is the constructor that takes care of initializing the spatial and temporal limits of the trip, which are represented with an *IntervalTraversal* object. The max speed is also set from the *halifax.csv* or *mcdv.csv* files, which describe the different motor configurations.
2. Then the *buildFirstSchedule* method is called. Its role is to build a schedule without taking the weather into account to be used as a starting point.
    a. The mission tasks defined in the request are then transformed into the necessary *ToyTask* objects and these are given, in list form, to the *Scheduler* to produce a *SchedulingResult*.
    b. A null check is performed on this result, due to the possibility that the tasks are over-constrained or that the trip is impossible to schedule. In this case, the appropriate error flags are raised and the method is stopped. These flags will be picked up by the *RequestProcessor* and sent to the appropriate message producer if need be.
    c. Finally, the schedule is produced and checked for success.

3. After the creation of this first schedule, it becomes possible to associate a certain time to our *Step* locations (see 3.3.1 item 3) and hence to check the weather at the corresponding points and times. If the task's spatial interval only contains points where weather conditions are compatible with that task, then the scheduling process is over, since a weather-viable solution has been found.
4. In the case of a weather incompatible schedule, other iterations are performed.
    a. In these other iterations, the weather from the preceding iteration is taken and used to redefine the new spatial availability of the tasks being scheduled. (If a tempest prevents a helicopter launch, then the space occupied by the tempest is removed from the spatial availability of this task.) This way, the new schedule can refrain from setting tasks in space and time where the weather is incompatible. While this offers no definitive guarantee of success, this process is repeated 5 times before giving up if no successful schedule is found.

3.3.3 FuelOptimizerProcessor

1. The processor object is created from a database access object (Given from Spring's configuration), from the ship type and from the ActiveMQ message producer (in order to report errors).
2. The optimize function is then called.
    a. The first step is the initialization, where the necessary motor configuration file is read and parsed in order to create a list containing the different engine configurations. Each engine configuration has a corresponding speed and resistance in calm water, as well as associated fuel consumption.
    b. The different "Move" actions (Not the tasks) are then split into stable weather zones when the Mahalanobis distance between environmental conditions in two adjacent areas is greater than a certain value. This cutoff value can be modified in the *application.properties* file. Since the Mahalanobis distance has a possibility to fail (some matrix operations used can result in null values), a standard deviation method is written as a backup solution.
    c. Each of these stable weather (and constant ship heading) "Move" actions is then optimized to minimize fuel consumption. The variables used are the amount of time spent in each engine configuration. The selected constraints are described in more details in the *FuelOptimizer* section but consist of the following:
        1. The sum of the time in each motor configuration must be equal or less than the action's time.
        2. The sum of the product of the time and the effective speed in a motor configuration must be greater or equal to the distance covered by the action.
    d. The output of the simplex solver, which takes care of the optimization, is the time that the ship should spend in each motor configuration so as to traverse the route interval using as little fuel as possible.

3.4 User/administrator guide

The AutoScheduler service receives a *SchedulerRequest* with its accompanying *RequestID* and gives back a *ScheduleResponse* as an answer tagged with the same *RequestID* for identification purposes. This section describes how to start and use the application.

To start the Scheduler, first make sure your *application.properties* file reflects the values you wish to use. Then the program can be compiled by using:

```
cd C:\your_installation_dir\code\Scheduler\COATScheduler
mvn clean install
```

provided that the appropriate dependencies have already been set (See Section 5). Starting the service can then be done with

```
cd target
java -jar COAT-Scheduler-0.0.1-SNAPSHOT.jar
```

If a user desires to restart the application with a different configuration but without recompiling, the properties to be modified can be loaded as environment variables by adding them in the run command (in the same subdirectory as above), like so (Do not forget the D right after the -):

```
java -jar -Dvariable.to.be.modified=newValue COAT-Scheduler-0.0.1-
                                SNAPSHOT.jar
```

See Annex C for a list of available parameters for this module.

Designed to be a COA-T implemented system, the *AutoScheduler* is a service listening on the ActiveMQ request topic until an appropriate message (Of *JMSType=SchedulerRequest*) arrives. Until such a message arrives, the service is essentially in a waiting mode, only sending *HeartBeats* to the COA-T infrastructure in order to advertise its existence.

SchedulerRequest messages can be created and sent from the *SchedulerRequester* application that was built as part of this call-up. However, modifications to a request built through this application must be made through a Java IDE. Of course, any ActiveMQ connected applications could send to the *C2.REQUESTS* topic the correct JSON formatted message that will instruct the Scheduler to build a *SchedulerRequest* object. The exact format of such a *SchedulerRequest* is available in Annex A. A typical request contains waypoints describing the route of the ship, trajectory type, ship type, start time and end time, whether or not fuel optimization is to be applied and the tasks to perform along the route. The task data model is also defined in Annex A.

In the current configuration (Which follows the AMQ Design Document provided as a GFI), the Scheduler's AMQ listener watches the *C2.REQUESTS* Topic in order to receive the requests and gives the answers back on *C2.REQUESTS_DATA* as a JSON formatted message.

A brief description of an example of the *COATScheduler* service in action can be seen next:

Starting the COATScheduler

1. From *COATscheduler/src/main/java/ca/drdc/COATScheduler*, start the *CoatSchedulerApplication.java* main.
2. The service is started, periodically sending HeartBeat messages.
3. The service is now waiting for a Schedule Request.

Sending the SchedulerRequest

1. From *Requester/src/main/java/ca/drdc/c2sim/schedulerequester*, start the *SchedulerequesterApplication.*
2. This builds *MissionTasks* and a *SchedulerRequest* object that are sent through ActiveMQ.

Processing the SchedulerRequest by the COATScheduler

1. A listener is awoken by the arrival of a message and processes the request.
2. From the request, the zone of operations is defined and a 1D line is built from the waypoints.
3. Weather Data is requested, for the area of operations and for the necessary timeline.
4. Each task's 2D (*LatLon*) spatial requirements are transformed into 1D requirements for the Scheduler. The 1D line describing the route is split into a succession of *Step* objects. These steps are now described by their 1D position and their LatLon position.
5. The *SchedulerProcessor* is called and a first schedule is built. For every step in the voyage, the expected time of arrival at this point is calculated.
6. The weather validity of this schedule is verified by checking the expected weather for every point where a task is scheduled against this task's constraints.
7. If the schedule is valid, it is sent as a result. If not, the zones where the weather prevents a task from being completed are removed from the task's spatial intervals. Four more iterations of this process are done if unsuccessful.

3.5 Developer guide

3.5.1 Generic development information

This module and all the other ones were developed in Java 8 using Spring and Maven tools. Consequently, each main module directory contains a *pom.xml* file containing all the configuration and dependencies necessary for successfully compiling that module. The version of the module is also being set in the *pom.xml* file and is set to *0.0.1-SNAPSHOT* and should be changed if the code is modified in a new release.

When compiling for the first time, all dependencies (Java libraries) are downloaded from the Internet and stored in an repository tree located in the user home directory name .m2. Should the user need to compile the software offline (without Internet access), then the repository tree must be delivered and installed in the user home manually so the compilation can be completed.

3.5.2 Data Format

Since the *COATScheduler* is meant to be integrated into COA-T, any change made to the message structure used by this application needs to be reflected in the code. For example, the Scheduler expects a very precise format for the Weather Requests and their answers and therefore any deviation from this expectation will result in incomplete data (For the exact expected structure, please consult Annex A).

3.5.3 Route building

The itinerary, the object describing our list of steps, goes through several rebuilds during the scheduling process. It is therefore useful to explain the process here.

1. The *SchedulerRequests* gives a list of *LatLon* waypoints. The *RequestProcessor* takes these multiple points and joins them to form a 1D line composed of small *Steps,* each of which describes a spatial interval on the route.
2. This list of *Steps* is then entered into the DRDC-*AutoScheduler* and a *ScheduleResult* is outputted.
3. This *ScheduleResult* is quickly parsed into *ScheduleAction* objects. Each of these indicates whether it's a move action or a task and has a spatial and a temporal interval. These intervals indicate where and when the *ScheduleAction* is accomplished.
4. These *ScheduleActions* are still in a 1D format, which is not practical for a navigation officer or for our algorithms. Therefore, these actions are then split into parts over which the original route has constant heading. This is necessary because *ScheduleActions* can sometimes extend over several legs of the route.
5. If fuel optimization is requested, the *ScheduleActions* that are only moving actions are then split again into parts of nearly-constant weather. Nearly-constant weather is defined with reference to a statistical distance measure (namely the Mahalanobis distance), which measures the similarity of two vectors of environmental conditions. The more similar two environmental prediction vectors get to one another the smaller the Mahalanobis distance between them will be. The parameters used to split the route in this way are as follows: wave height, wave period, wave heading, horizontal wind speed, vertical wind speed, horizontal current and vertical current.
6. This final list of *ScheduleActions* is then JSON-serialised and included into a *ScheduleResponse* object, also JSON-serialised. This response is then sent to *C2.REQUESTS_DATA* with its accompanying *RequestID*, *ClientID* and other necessary header properties.

3.5.4 Fuel optimization

After a schedule is built, the output is a list of *ScheduleActions*. These last could be tasks or simply "Move" actions, where the ship moves through a spatial interval over a given time interval. The goal of the fuel optimizer module is to take these "Move" actions and find the set of motor configurations that will consume the least amount of fuel over the leg. To do this, a couple of steps are taken, as described below.

3.5.4.1 Split the "Move" action into smaller homogenous parts

Splitting a move action is a balancing act between obtaining small enough parts to get a better optimization process by ensuring nearly-constant conditions and yet big enough ones so that the crew does not spend all its time switching motor configurations. To satisfy both these constraints, a process has been designed:

1. A length of 100 nautical miles, starting from the spatial interval min, is first selected.
2. The average environment conditions of the next 50 miles are computed. If the Mahalanobis distance between the conditions of this part and those on the next is smaller

than the cutoff threshold, the next 50 miles are added to the previous interval. Otherwise, a new set is started.

3. The previous step is repeated, as long as the remaining spatial interval is longer than 50 miles. Once less than 50 miles remain, these are simply added to the working set.

This simple process allows the module to get a certain level of homogeneity for the intervals while keeping them at least 50 miles long.

3.5.4.2 Optimize over the sum of smaller parts

Fuel optimization is performed by selecting a sequence of motor configurations which will ensure that the minimum quantity of fuel is used while still travelling the entirety of the transit in the permitted time. Since the previous step broke the transit actions into smaller legs, it would be possible to perform optimization on each these legs, one by one, as seen in Figure 1.a). This approach would need to select motor configurations that would respect Δt and Δx, find the lowest-cost solution and repeat for every other sub leg. Even more, if a leg from a to b is split into two sub legs, both these sub legs will need to have the same average speed as the main leg which could prove very fuel expensive if there is a tempest in the first sub leg.

In the solution described in Figure 1.b), the whole leg has been split in such a way as to maintain the same average speed on each sub leg is the same which could translate in very high fuel costs or even schedules that are impossible to set (If the maximum effective speed is lower than the required one).

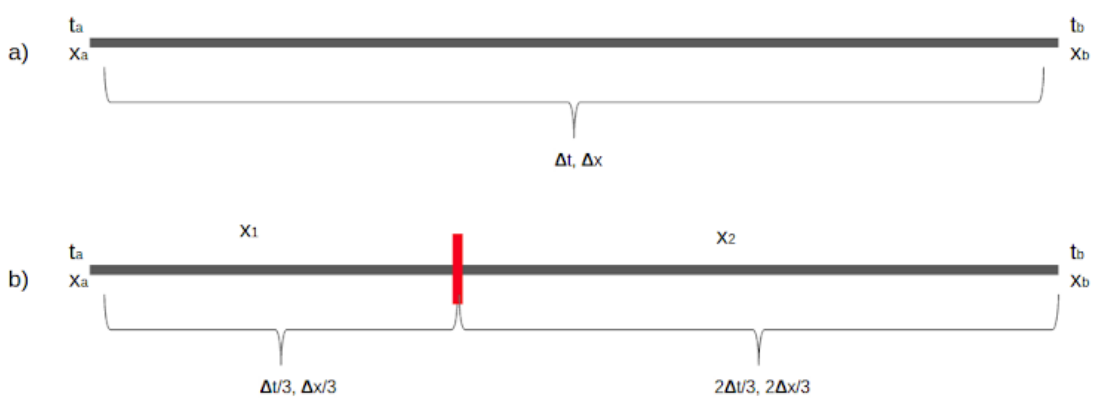


Figure 1: a) represents the leg before the splitting. It would be possible to perform optimization over the average wave conditions of the whole trip but this would be much less effective. b) a split has been performed at the third of the spatial interval for simplicity. A possible solution for optimization is to split the leg in such a way as to maintain the same average speed before the new boundary (shown in red) as after it.

However, if the optimizer is allowed to go slower in the first sub leg and is able to pick up the slack in the second sub leg where the weather is more clement, then fuel gains can be accomplished. To do this, the constraint describing that each sub-leg time of arrival must be respected is replaced by one saying that the total time of all sub legs must be respected. This approach was selected and translated into mathematical equations where the individual Δx_i must be respected but only the total Δt is considered as a constraint.

Therefore, our final constraints for the situation can be described as such for a system with two sub-legs, where *i* iterates over the number of legs and *y* over the different motor configurations available. *t* represents the time spent on the spatial interval *i*, v_{eff} represents the effective speed of a motor configuration given the weather and *F* the fuel consumption of a particular motor configuration.

$$\sum_i \sum_y t_{i\text{-}MCy} \leq \Delta t_{ab} \qquad \sum_y t_{1\text{-}MCy} v_{eff\ MCy} \geq x_1 \qquad Min(\sum_i \sum_y F_{MCy} t_{i\text{-}MCy})$$

$$\sum_y t_{2\text{-}MCy} v_{eff\ MCy} \geq x_2$$

These three columns describe respectively that:

1. The sum of times spent on each leg and in each motor configuration must be lower than the total allowed time by the schedule
2. The time spent in each of the motor configuration during the first leg multiplied by the motor configuration speed must be greater or equal to the length of the first leg. The same constraint applies to the second leg.
3. What is being minimized is the total fuel consumption on each leg. The consumption on a given leg is obtained by summing up the product of the time spent in each motor configuration with the associated fuel consumption rate, over all *y*.

3.5.4.3 Using the Simplex algorithm

The simplex library that was used for this call-up, Apache Commons Math3, is a very well-documented way to define constraints and solve problems. The first step is to define the linear objective function, which in our case is an array of fuel consumption per hour for every motor setting. This array represents the coefficient that when multiplied to the time spent in each motor is the value to be minimized.

The second step is to define the constraints, under the same format of coefficients multiplying the array of time spent on the configurations. The final step is to simply create a *SimplexSolver* object and call its *optimize* function and give it its necessary inputs: the objective function, the array of constraint arrays, its goal type (minimize or maximize).

4 ShipMo7 Service

4.1 Requirements

The call-up SOW provided this initial requirement: Integrate the existing Shipmo7 software into the COA-T to be used as a service. This goal has been achieved with slight modifications.

4.2 Solution Overview

ShipMo7 is an application used by the Canadian Navy and DRDC in order to predict ship motions in various wave conditions. Here the interest is mainly in its ability to compute the added water resistance in irregular waves. It was therefore deemed important for this system to be included in the COA-T system. However, the .exe encapsulation of the software, the fact that the source code was in Fortran, as well as the size of the inputs all suggested it would be too challenging to integrate the code itself.

It was, instead, deemed easier and better to pre-run Shipmo7 under every possible combination of wave heights, period, direction and ship speed for both the Halifax type vessels and the MCDVs. The resulting data from these simulations was then loaded into a PostgreSQL database, which can be accessed either by direct SQL requests or through the COA-T service. The output parameters that were deemed of interest were the added resistance and the frequencies of keel emergence, deck wetness and MII.

4.3 Design

In this section, we will overview the main logic behind the processes in getting a Shipmo7 response from a COA-T request. A request is built from five elements and a *requestID* (In the request's properties):

1. Vessel type (Halifax or MCDV)
2. Wave Height (in meters)
3. Wave period (in seconds)
4. Ship's speed (in knots)
5. Sea Heading (In degrees, relative to the ship)
6. The *RequestID*, a uniquely generated code used to match a request and its answer

The service then selects the appropriate table (Based on the ship type) and builds an SQL request. The parameters from the request are adjusted to match the closest found in the database (i.e.: an input value of 3.15 would be treated as a 3.2). The SQL request is then sent as a statement and the database answer is given back in an answer message. The answer contains the *requestID* in its properties, the parameters used as database inputs (for reference) and, of course, the irregular wave resistance (in Newtons), keel-emergence and deck wetness per hour and MII per minute.

4.4 User/administrator guide

This program can be started the same way as the *COATScheduler* with a simple

```
java -jar shipmo7-0.0.1-SNAPSHOT.jar
```

The parameters to be modified are however different, besides the topic properties and ActiveMQ connection information. See Annex C for a list of available parameters for this module.

The general logic of the *ShipMo* service stays the same: listeners are set up and await a request on the appropriate topic. This request is parsed and a result is given. The following gives a more detailed view of this process:

After startup of the ShipMo service:

1. From COAT*shipmo7/shipmo7/src/main/java/ca/drdc/shipmo7*, start the *Shipmo7Application.java* main (using Eclipse for example).
2. The service is started, periodically sending *HeartBeat* messages and its listeners are waiting.

Sending a ShipMoRequest

1. From *shipmo7Requester/src/main/java/ca/drdc/shipmo7Requester*, start the *Shipmo7RequesterApplication.java* main.
2. This builds a simple *ShipMoRequest* and sends it to the *ShipMo* service.

After detection of the incoming *ShipMoRequest*

1. The JSON request is parsed and a postgreSQL request is built. The database query is adapted to fit the precision and structure of the databases content.
2. When the query's answer is given, the information is then sent as a *ShipMoResponse* message.

4.5 Building the database

The *shipmodata* database should not be rebuilt unless issues become apparent. Perhaps the files given to OODA will be found to contain errors. Perhapsthe database will become corrupted or inconsistencies could be discovered. For reasons like these, the code folders delivered, namely *shipmoDBbuilder*, contain two projects: one for Halifax vessels and the other for MCDVs.

A *DBbuilder* project is built to function on four threads (For four cores) in order to save time. Each of these threads will need a folder containing a copy of ShipMo.exe with the name SHIPMO1, SHIPMO2, 3 and 4.

When the program starts, four *ShipMoThread* are built and are assigned a *shipmo7* folder, a *ShipMoRunner* responsible for launching cmd prompts, an *InputBuilder* for creating the necessary .inp files for ShipMo and an *OutputExtractor* for reading the .out files. The final step is to take the information parsed by the *OutputExtractor* and save it into the database.

These steps are repeated by the *ShipMoThread* over a range of wave heights, wave periods, sea headings and ship speeds. In order to parallelize the calculations, each thread takes a quarter of the available waveheights and iterates completely over the other parameters. Further

parallelization is possible by splitting the range of wave heights again and distributing it over multiple computers, as long as these are connected to the database.

5 Obstacle Avoidance Service

5.1 Requirements

The call-up SOW provided this initial requirement: Implement a shortest route finding algorithm and integrate this into the COA-T to be used as a service.

The route must avoid depths shallower than the ship's draught. The bathymetric data used comes from S57 charts provided by the TA.

5.2 Solution Overview

5.2.1 Input

To build an application that satisfies this need, the *ShortestPath* java project was created, where an AMQ listener expects a *ShortestPathRequest* JSON formatted object, which contains a start point (as a *LonLat*), an end point and the ship's draught in meters.

Two optional parameters also exist, which modify the grid used to create more precise itineraries, which will take longer, or more imprecise ones, which will be quicker. The parameters work by changing how much you increment in size between two points on the grid: which decimal place you modify and by how much you modify it.

The first of these parameters is called *decimal*. It is an integer describing the position of the last decimal increment to be kept in the LatLon grid. The second parameter is the *precision*, another integer, which describes the value of the increment between different points on the grid. Combining *precision* with the number of the increment at the position described by *decimal* will give the distance between two points on the grid. In order to limit rounding errors given by the use of *double*, the value of *precision* is limited to 1 or 5.

Tests performed by OODA showed that the best combination when considering time and performance was *decimal*=2 and *precision*=1. These parameters were therefore selected as the default values but for greater flexibility to DRDC, a particular request can ask for different values.

5.2.2 Processing

When the ShortestPath application starts, it creates its *EnvironmentData*, where the different polygons describing the depth areas stored in the S57 maps are loaded into memory. When a *ShortestPath* request is received, the service will verify that the start and end points are valid and

will use the A* algorithm to build a series of nodes towards the end point. To check if a node is accessible or not, queries are made into the *EnvironmentData* to see in which bathymetric polygons the node is contained and hence what the depth over the node would be. When the final node contains the end point, the process is finished.

In this type of path optimization, steps along the route are chosen by considering which possible next point would take us close to the objective (Heuristic being the term used to describe this value) and by also minimizing the cost (The length of travelling from a point to its neighbour), see section 5.2.2.

5.2.3 Output

The response that is sent out is a *ShortestPathResponse*, an object containing a list of *Point* objects describing the itinerary as *double*[] containing Longitudes and Latitudes.

For convenience' sake, CSV files containing this list of *LonLat* are also created when the *ShortestPathRequester* receives a response. Such a CSV file can then be input into Google's MyMaps service to create a quick visual representation of the itinerary.

5.3 Design

In this section, the main logic and algorithmic design will be summarized. The *ShortestPathRequest* is built from 6 configurable properties and a *RequestID*.

1. **start**: Starting point (long, lat)
2. **end**: End point (long, lat)
3. **maxdraft**: Maximum depth / ship draught (in meters)
4. **decimal:** The position of the last decimal which will be incremented between two points on the grid.
5. **precision**: The value of the last decimal which will be incremented between two points on the grid.
6. **speed**: the speed of the ship(not used for the moment)
7. The *RequestID*, a uniquely generated code used to match a request and its answer

The service is built in two parts in order to benefit from malleability. The first part is the way the depth information is gathered.

5.3.1 DepthService

The first part of this section concerns the way depth information is gathered through S57 charts. Also, the *DepthService* has been built with the possibility of being replaced by an alternative depth query system, should the S57 maps prove inadequate for this purpose. If another query system is built, it would need to implement two interface-dependent methods:

1. *isValidPoint* taking a longitude and a latitude as input and returning -1 if the point is on earth or the data is not available and the minimum depth in absolute value if the data is present.

2. *depthRange* taking a longitude and a latitude as input and returning an array containing the minimum depth in first position and maximum depth in second.

Care also needs to be taken since some functions need to be thread-safe. The code documents some synchronized function.

Spring-Boot[2], an application framework for Java, permits users to select which depth service to use via a parameter in the .properties file (see Annex C). The new service must implement DepthService, must be a Spring component and must have the *ConditionalOnProperty(name = "properties.depth.calculator", havingValue = "name of the service"*). For this call-up, an S57 chart depth query system is given. Each S57 chart used must come with a DEPARE layer and an extent, which correspond the bounding box of the chart. It uses the GDAL library to extract depth areas (DEPARE layer) as polygons. Each polygon can stand alone or can have other polygons inside, which correspond to other depth areas. Each main polygon is associated with a depth range, which corresponds to the minimum and maximum depth contained in the area. The polygons are extracted from the S57 charts given by the folder path in the *.properties* file.

The implementations of the two methods are as follows:

1. *isValidPoint* determines which *extent* (an *extent* is a rectangle containing depth polygons) the input point lies in and iterates over the polygons of this *extent*. If a point is in a main polygon, but also part of a sub-polygon it is outside the main, as the smaller ones represent zones of exclusion. If the point is in no polygons or *extents*, a value of -1 is returned. This is also the case if the point is on land. The minimum value is returned otherwise. For example, if a particular area has depth values between 40 and 50 meters, the value of 40 will be returned since a ship with a 45 meters draft could scrape the bottom in some parts. It is to be noted that if a point is in several *extents*, the minimum depth taken in consideration is the maximum of the minimum possible depths, since a higher minimum is associated with higher precision, see Figure 2.

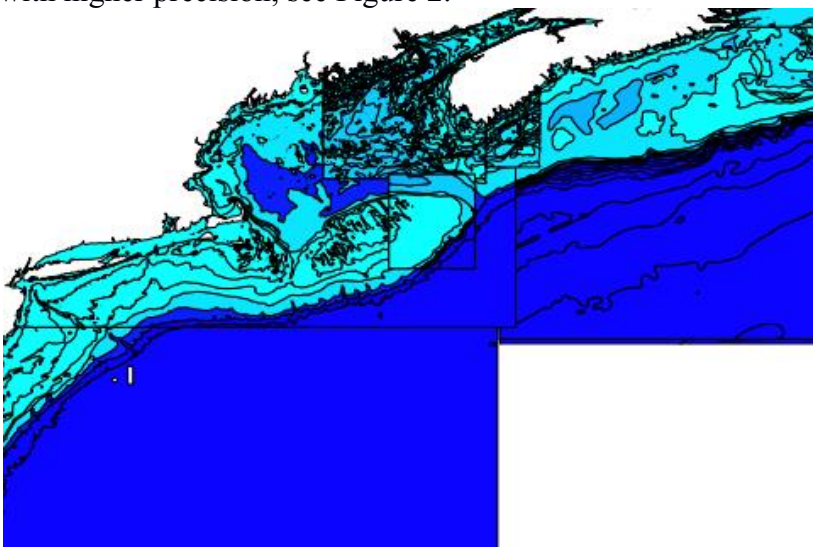


Figure 2: Here we can see on smaller, more precise charts that the maximum depth is less than on a bigger, less precise chart.

[2] https://en.wikipedia.org/wiki/Spring_Framework

2. *depthRange* takes the same procedure to return the interval rather than just the minimum, and returns -1 if not available or on land. If the point is in more than one *extent,* the maximum of the minimum and the minimum of the maximum are returned.

5.3.2 RequestProcessor

The second part is the custom implementation of the A* algorithm. It is too memory consuming to have a mesh of the earth with precomputed depths with a reasonable precision. An alternate solution pursued by OODA is to only load the nodes (Points that have been explored as valid or not) as they are needed.

As mentioned before, two important values when using A* are the heuristic and the cost. The heuristic is the great circle distance between the node and the end, and the cost is the great circle distance between the node and its neighbours. This heuristic is monotonic and admissible considering the cost. With those constraints, an implementation of the algorithm can be done with a *closedSet* method, which stipulates that if a node has been expanded (Meaning if its neighbours have been explored as valid points where the depth is sufficient or not) it won't be expanded again and the algorithm will lead to an optimal path. The implementation is as follows:

5.2.2.1 Point Object

Each observed node of the grid is represented by a Point object:

1. Coordinates, an array containing the longitude and latitude
2. Depth, containing the depth extracted by the service
3. Parent, representing from which node it is from
4. Cost, the cost to get from the start node to this node
5. Heuristic, the optimal cost from this node to the end node
6. *fvalue*, the sum of the cost and heuristic

The A* algorithm begins with the starting node as follows:

1. The starting and end points are checked for validity

2. The *openList*, *closedList*, *existingList* and *nonValidList* are created. The *openList* consists of the nodes seen by expansion, but not expanded yet; the *closedList* contains the nodes expanded; the *existingList* contains the nodes that have been created and the *nonValidList* contains the nodes that are not valid. The starting node is created and put in the *closedList* and *existingList*. The node is expanded, which means all its valid neighbours are put in the *openList* and in the *existingList*.

3. While the *openList* is not empty, the node with the least cost + heuristic (f value) is put in the *closedList* and its neighbours in the *openList*.

4. When a node is expanded, the neighbours of that node that could be accessed are determined. For each neighbour, the method checks if the node has already been created. If it exists and is in the *openList*, it checks to see if the path to the current node through that neighbour is shorter than the already existing way to this node. If so, it changes the parent

node to the expanded node and the cost to the new cost. If the node doesn't already exist, it checks if it is a valid node and if so, creates it. If it is an invalid node, the coordinates are stored and the node will never be tested again, as it will be sent into the *nonValidList*.

5. When the expanded node is the end node, the algorithm terminates and sends a list of nodes representing the path. This list of nodes is constructed by following the end node to its parent and then that parent to its parent and so on, all the way back to the start node.

Since the mesh is on a sphere, the way to visit the graph is not typical. The first tests were made with a conventional 8 neighbour pattern.

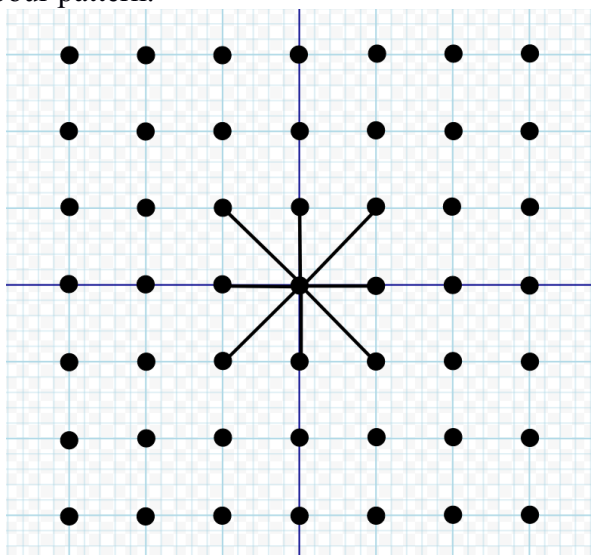


Figure 3: 8-neighbours pattern of the grid.

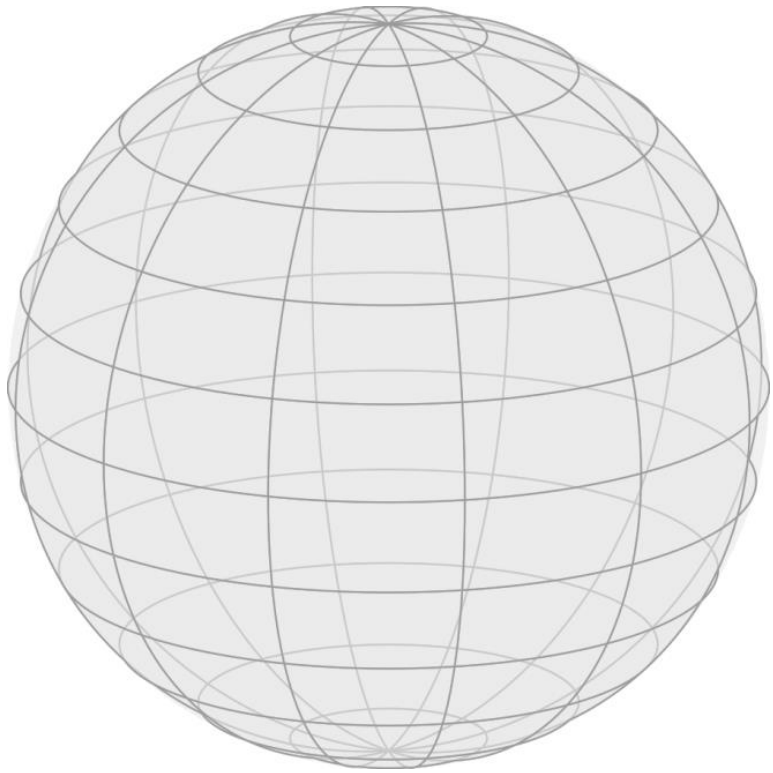
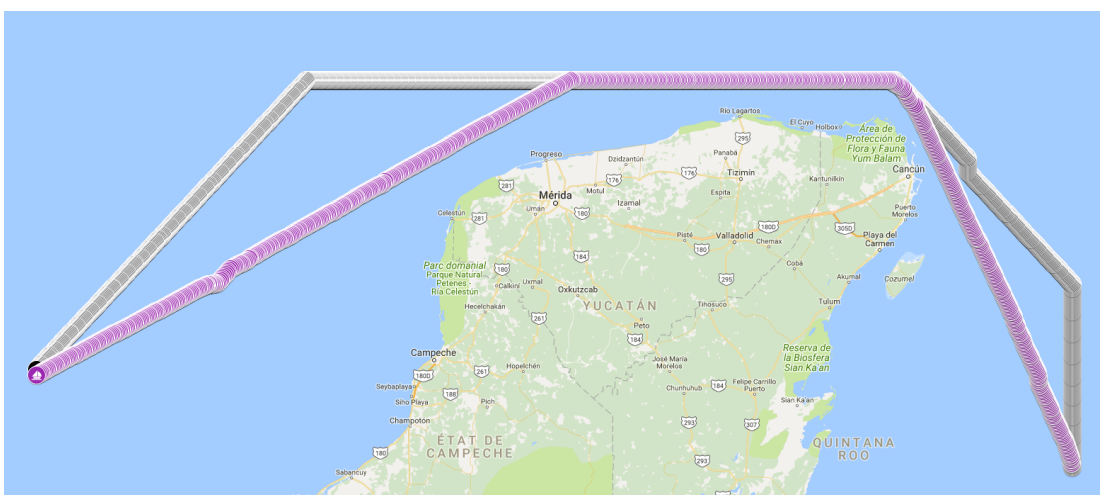


Figure 4: The mesh of a sphere where latitude and longitude have constant increments.

Since a mesh on a sphere looks like Figure 4, the length between two longitudes with same latitude is not constant. This creates a mesh grid with more of a trapezoidal shape. The solutions on this grid create paths that are not optimal at first sight.



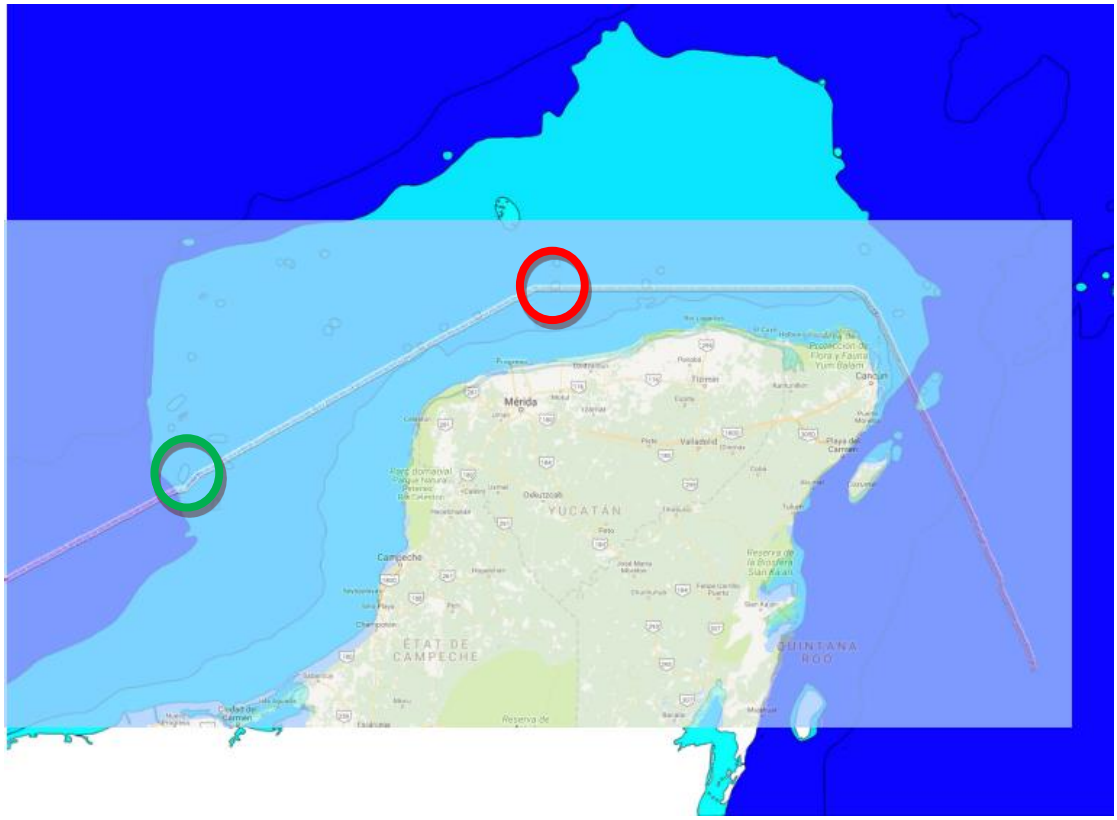


Figure 5: **Top**: The grey path with a 45-degree angle is the solution taking only the 8 first neighbours. This form is optimal for the grid because making a complete x-step costs less at higher latitude which, in this grid, is more optimal than a staircase scenario. The smoothest purple path is the result of taking a 16 neighbours pattern. **Bottom**: The second is an overlay of the S57 map used and the purple trajectory. The kink in the green circle is explained by a small plateau blocking the straight path. In the red circle, it seems the itinerary is going through part of another plateau. The size of the spot being jumped over was under ≈500 meters, our level of precision. This can be avoided by higher precision levels but at the cost of computing time.

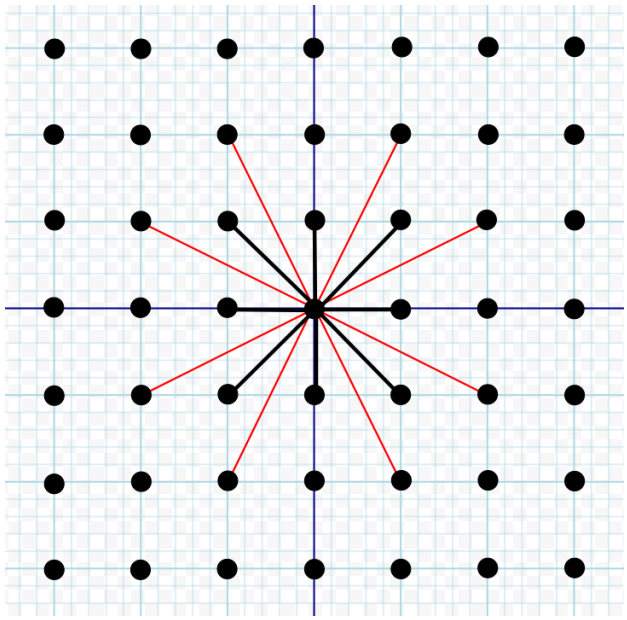


Figure 6: 16 neighbours configuration taking second order neighbours and neglecting the ones that can be accessed by only two operations.

The star pattern (Figure 6) has been selected since it gives overall better results in minimising the path cost. By adding wider angle range, it results in a more straightforward path, with a better cost. It should be noted that the wider the star pattern, the less precise the path will be, in the sense that the arms become longer and therefore could possibly cross land[3]. Longer computation time is also a factor since more nodes are created. In the specific case where the Figure 6 star pattern is taken, both the number of nodes expanded and the cost of the optimal path of the grid are minimized. After multiple tests, the number of closest neighbors to explore has definitely been set to 16, since the augmentation of neighbours explored did not result in significant better solutions over the possible precision loss of longer arms. This addition gives trajectories that match those of a similar ship path-finding service, gives comparable computing times and the points are still very close, meaning that the risk of crossing land/unavailable areas remains very low.

5.3.3 Comparative results

"What DRDC is looking for is similar to capability already commercially available in app form, such as
https://www.navionics.com/usa/charts/features/dock-to-dock-autorouting."

Here is a quick test to compare results and performance between what DRDC is expecting and what OODA provides:

[3] It becomes very expensive very quickly to check against every pair of points forming every side of every polygons if they are crossed. It would be better to augment the precision if issues are raised.

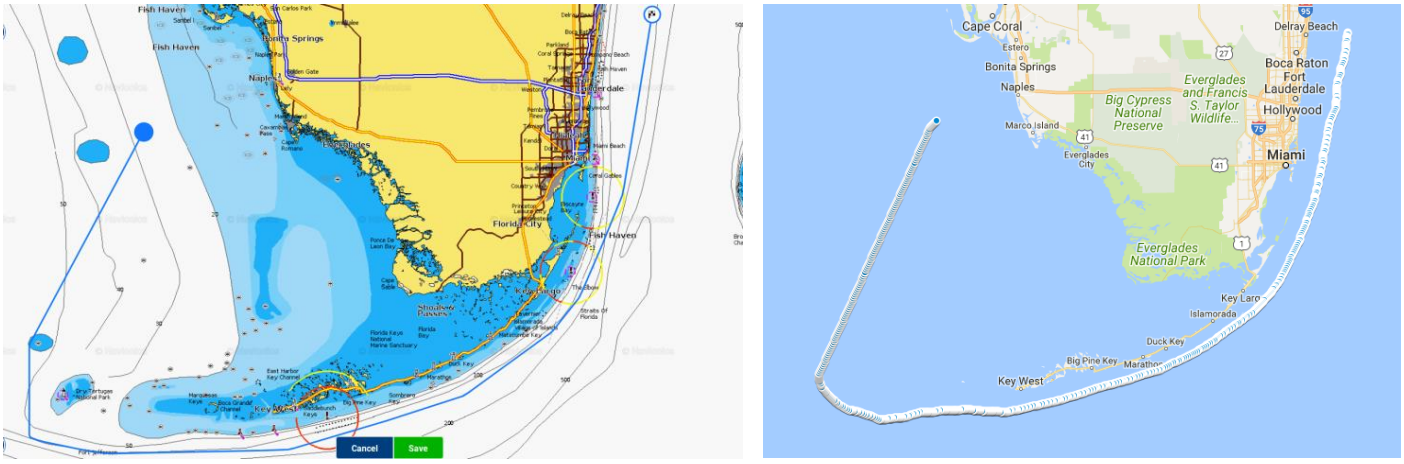


Figure 7: The first path is the result given by the Navionics software with a compute time of 1 minute 20 seconds and the second is the path computed by the shortest-path service in 1 minute 52 seconds. A faster compute time could be achieved with another depth service since looking in an S57 chart is very CPU demanding.

5.4 User/administrator guide

Since this application critically relies on the S57 map files, make sure that the path to the s57files directory is set in the *application.properties* or through input commands when starting the jar by adding *–Dproperties.s57directory.path=YourPathComesHere* . As a last note before the steps, it is important to make sure GDAL is installed. Instructions are provided by the GFI DRDC document "GDAL Install Guide"

1. This program can be started the same way as the *COATScheduler* with the command
1.2 cd ShortestPath/COAT-ShortestPath/target
1.3 *java -jar COAT-ShortestPath-0.0.1-SNAPSHOT.jar*.

The parameters to be modified are however different, besides the topic properties and ActiveMQ connection information. See Annex C for a list of available parameters for this module.

2. The general logic of the COAT-*ShortestPath* service stays the same: listeners are set up and await a request on the appropriate topic. This request is parsed and a result is given. The following gives a more detailed view of this process:

After startup of the ShortestPath service:

1. From COAT-*shortestPath/src/main/java/ca/drdc/COATshortestPath*, start the Coat*ShortestPathApplication.java* main (using Eclipse for example)
2. The service is started, periodically sending *HeartBeat* messages and its listeners are waiting.

Sending a ShortestPathRequest

1. From COAT-*shortestPath-Requester/src/main/java/ca/drdc/COATShortestPathRequester*, start the CoatShortestPathRequesterApplication.java main (using Eclipse for example). This builds a simple *ShortestPathRequest* and sends it to the *ShortestPath* service.

After detection of the incoming ShortestPathRequest

The JSON request is parsed and sent to the processor. The A* algorithm then finds the shortest path between the starting point and the end point. The starting point and end point are adapted to fit the precision of the grid.

When an optimal path has been found, the information (a list of lat/long waypoints) is sent as a *ShortestPathResponse* message.

6 Conclusion

6.1 Lessons Learned

6.1.1 COA-T Data Model Modifications

Changing the structure of COA-T means revisions and uncertainties. COA-T message protocols were still being modified at the beginning of this call-up or had been changed without the proper inclusion of information in the accompanying documents. Some work had to be redone.

In the same vein, the exact structure of Weather Messages still seems to be flexible (Since the weather service has not yet been created), while the current application expects an exact structure. This could possibly lead to problems in the future.

6.1.2 S57 Map limitations

The depth accuracy is somewhat limited when using S57 map. The depth intervals between isolines are not uniform throughout the map. Also, the accuracy is lacking in the most important region, which is between the coastline and the first depth isoline (i.e. 0-18m), much too vague for drawing a safe trajectory which takes into account the draught of the ship.

6.2 Recommendations and future development

6.2.1 Replace the Requesters

The requesters have been created as both placeholders until DRDC chooses what would be the best way to interact with these COA-T services and also as a way to test the request creation and response reception. The creation of a GUI facilitating the creation and display of both requests and responses would be incredibly useful.

6.2.2 Obstacle avoidance service

Weather information could be added in the cost function to get faster path with weather concern (it would be possible that the present heuristic would not be valid in this scenario.)

Several requests could be made to generate a *heatMap* of the most possible path considering a starting region of interest.

Since each request is independent from one another, more than one request could be processed at a time and with a cluster of computers, several requests could be processed at the same time resulting in faster results, if an area of starting or ending points were to be tested.

The depth service produced in call up 18 could be added to add precision and efficiency since a lot of depth requests are made and the format of S57 is not as precise.

Annex A: ActiveMQ Message Structure

Since our services implement a lot of new message types that will be sent through ActiveMQ, it is important to both notify future developers on the structure that is used and inform administrators so the proper documentation can be updated.

SYSTEM Topics and Messages

01DB-PSA Design Document – ActiveMQ still formulates that the message in the SYSTEM topic and its subtopics must be in a MapMessage format. It is important to modify this document to reflect the current design choice of putting the data in a TextMessage with a JSON formatted body.

Requests

The messages requesting a schedule to the COAT-Scheduler shall follow this format. After receiving the schedule request as a REQUEST_SCHEDULE JMSType message, the service will process it and respond with a REQUEST_SCHEDULE_DATA JMSType message, both going through the C2.REQUESTS Subtopic.

Table 1: REQUEST_SCHEDULE type message

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
Header	**JMSType**	**"REQUEST_SCHEDULE"**	**String**	
Properties	**RequestID**	**The ID of the request**	**String**	**The unique ID of this request – this field should not be blank. Generated as a GUID to ensure uniqueness.**
	ClientID	**The ID of the requester**	**String**	**The identifier of who's sending the request. (Since our requester service is not a final COA-T product, this is currently omitted.)**
	APP_NAME	**Name of the sending application**	**String**	**Which application sent data ("AUTOSCHEDULER-REQUESTER" for our example requester)**
Body (JSON text string, containing these	**Route**	**Array of lat/long describing the waypoints**	**Array of Coordinates (lat/lon double combination)**	

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
values)	**RouteType**	**Type of trajectory between waypoints**	**String**	**This string shall either be "GREAT_CIRCLE" or "RHUMB_LINE"**
	BoatType	**The type of ship requesting the schedule**	**String**	**This string shall either be "HALIFAX" or "MCDV"**
	StartTime	**The starting time the requested schedule**	**long**	**ms since Epoch.**
	EndTime	**The ending time for the requested schedule**	**long**	**ms since Epoch.**
	FuelOptimizationBool	**Whether to include or not fuel optimization in the Scheduling**	**Boolean**	
	Tasks	**A description of the tasks**	**Array containing Tasks**	**A Task JSON file is described in Table 2.**

Table 2: JSON formatted Task object

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
Body (JSON text string, containing these values)	**Name**	**The name of the task/Short description**	**String**	**Must be unique. A task that must be accomplished multiple times shall have #1, #2, etc. appended to it.**
	Duration	**How long the task shall take**	**Double**	**In hours.**
	SpatialConstraints	**Array of bounding boxes describing where the task can be done**	**Array of Coordinate duos, describing opposite corners of a bounding box**	**Box of constant lat/lon.**
	WindConstraint	**The max value for wind speed to do the task**	**Int**	**In knots.**

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
	WaveConstraint	**The max value for wave height to do the task**	**Int**	**In meters.**
	EffectiveSpeed	**The achievable speed along the route during the task**	**Int**	**In knots.**
	MotorConfig	**The motor configuration that represents fuel consumption during the task**	**Int**	**1-30 knots.**
	TaskPrecedence	**The name of a task which needs to be completed before this one can begin**	**String**	**Must be the precise name of the task.**

Using the format described above, we can easily build an example JSON message describing a Request with two mission tasks in an array (Presented later, for clarity.):

```
{
  "fuelOptBoolean": false,
  "route": [
    {
      "latitude": 0.0,
      "longitude": 0.0
    },
    {
      "latitude": 10.0,
      "longitude": 10.0
    },
    {
      "latitude": 0.0,
      "longitude": 15.0
    }
  ],
  "missiontasks": [SEE NEXT SECTION],
  "name": "todayRequest",
  "startTime": 1522071366812,
  "endTime": 1522356200000,
  "routeType": "GREAT_CIRCLE"
}
```

With the missiontasks array being represented by:

```
  [{
    "name": "A1",
    "duration": 5.25,
    "motorConfig": 15,
    "spatialDependency": [
      {
        "southWest": {
          "latitude": 2.0,
          "longitude": 2.0
        },
        "northEast": {
          "latitude": 3.0,
          "longitude": 3.0
        }
```

```
      },
      {
        "southWest": {
          "latitude": 7.0,
          "longitude": 7.0
        },
        "northEast": {
          "latitude": 11.0,
          "longitude": 11.0
        }
      },
      {
        "southWest": {
          "latitude": 5.0,
          "longitude": 1.0
        },
        "northEast": {
          "latitude": 11.0,
          "longitude": 7.0
        }
      }
    ],
    "waveConstraint": 6.0,
    "windConstraint": 40.0,
    "effectiveSpeed": 10

  },


  {
    "name": "A2",
    "duration": 2,
    "motorConfig": 5,
    "spatialDependency": [],
    "waveConstraint": 0,
    "effectiveSpeed": 5,
    "windConstraint": 0
  }]
```


Answer to Requests

After these requests for schedules are received, parsed and processed, the output must be communicated to the requester. The same subtopic, C2.REQUESTS, is chosen but with JMSType header of REQUEST_SCHEDULE_DATA instead. The RequestID value is reused in order to link the request to its answer.

Table 3: REQUEST_SCHEDULE_DATA type message

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
Header	**JMSType**	**"REQUEST_SCHEDULE_DATA"**	**String**	
Properties	**RequestID**	**The ID of the request**	**String**	**The unique ID of the request which triggered this response.**
	ClientID	**The ID of the Scheduler**	**String**	**The unique ID of this client**
	FuelOptimization	**If the fuelOptimization was activated for this autoschedule**	**Boolean**	

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
	APP_NAME	**Name of the sending application**	**String**	**Which application sent data (in this case should always be "AUTOSCHEDULER")**
Body (JSON text string, containing these values)	**success**	**Is the schedule doable?**	**Boolean**	
	fuelOptimizationSuccess	**Was the fuel Optimization done properly?**	**Boolean**	**This could be false if the weather was too severe, slowing down the ship so that the schedule is impossible to keep.**
	notes	**Description of the reasons making the schedule is impossible. Empty if schedule is possible**	**String**	**"NOT ENOUGH TIME DUE TO WEATHER" (Weather makes the end time is impossible), "SHIP NOT FAST ENOUGH" (Even without weather effects, the end time is impossible), "TASKS IMPOSSIBLE DUE TO WEATHER" (Wind/Wave constraint in the way of schedule)**
	ScheduledAction	**A description of when to start the different motor configs, the weather encountered, etc.**	**Array of ScheduledAction describing start time, motor configs, etc.**	**An example is given in the following section**

An overview of the rows present in the ScheduleOutput JSONArrayObject can be seen here:

Table 3: Elements of the ScheduleOutput JSON object present in the answer messages

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
ScheduleAction (JSON text string, containing these values)	**Waypoints**	**Beginning and end of this action part**	**LatLong[]**	
	Time Interval	**Beginning and end in time of this action part**	**long[]**	**Beginning and end in milliseconds since Epoch**
	Name	**Name of the task or "MOVE" if there is no task.**	**String**	
	IsTask	**If the action is a task or not**	**Boolean**	

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
	MotorConfiguration	**The motor configuration to be set**	**String**	**"GT14", "PDE11", etc.**
	EffectiveSpeed	**The expected speed to be achieved when weather effects are taken into account**	**Float**	
	AverageWind	**Weather to be expected during the task**	**Float**	
	MaxWind	**Weather to be expected during the task**	**Float**	
	AverageWave	**Weather to be expected during the task**	**Float**	
	MaxWave	**Weather to be expected during the task**	**Float**	
	DriftCorrectionVertical	**Expected drift in knots in the vertical direction**	**Float**	Due to current and wind
	DriftCorrectionHorizontal	**Expected drift in knots in the horizontal direction**	**Float**	Due to current and wind
	AverageHeading	**Ship heading in degrees**	**Float**	
	MII	**Motion Induced Interruptions per minute**	**Float**	
	KeelEmergence	**Number of emergence of the keel, per hour**	**Float**	
	DeckWetness	**Number of times the deck is washed with water, per hour**	**Float**	

Weather Requests

The expected COA-T weather service shall answer weather requests with every forecast available in the LatLon box described and in the time window described. These weather requests will be both posted and answered on the topic C2.INTERNAL_DATA.WEATHER .

Request messages:

Table 4: WeatherRequest message

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
Header	**JMSType**	**"WEATHER_REQUEST"**	**String**	
Properties	**RequestID**	**Unique ID of the request**	**String**	
	ClientID	**The ID of the Scheduler**	**String**	**The unique ID of this client**
	Variable	**The Weather type being requested**	**String**	**"WIND", "WAVE", "CURRENT"**
	APP_NAME	**Name of the sending application**	**String**	**Which application sent data (in this case should always be "AUTOSCHEDULER")**
Body *(JSON text string)*	**FORECAST_**	**TimeStamp describing the start of the timewindow requested**	**long**	**ms since Epoch.**
	TimeEnd	**TimeStamp describing the end of the timewindow requested**	**long**	**ms since Epoch.**
	MinLatitude	**Min Latitude for the box requested**	**Float**	
	MaxLatitude	**Max Latitude for the box requested**	**Float**	
	MinLongitude	**Min Longitude for the box requested**	**Float**	

	MaxLongitude	Max Longitude for the box requested	Float	

Answer to the requests:

The answer to a single request is the sum of multiple answer messages, the number of which depends on the number of forecasts contained into the interval (TimeStart-TimeEnd) from the request. All the messages that are created as an answer to this request will all have the same RequestID.

Table 5: WEATHER_DATA message

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
Header	JMSType	"WEATHER_DATA"	String	
Properties	RequestID	Unique ID of the request	String	
	Variable	The Weather type being requested	String	"WIND", "WAVE", "CURRENT"
	APP_NAME	Name of the sending application	String	Which application sent data (in this case should always be "WEATHER-SERVICE")
	Time	TimeStamp describing the moment of the forecast	long	ms since Epoch.
Body *(JSON text string, containing array of weathers containing these)*	Latitude	Latitude for which the described weather is valid	Float	
	Longitude	Longitude for which the described weather is valid	Float	
	Value1	Direction of the weather effect	Float	In degrees relative to North.
	Value2	Intensity of the weather effect	Float	Wind: WindSpeed, Wave: WaveHeight, Current: CurrentSpeed (m or m/s).
	Value3	Period of the weather effect, if existing	Float	In seconds.

ShipMo7 Module

ShipMo7 Requests:

Even though our database has a certain resolution associated (0.2 increments on wave periods, 5 degree increments on sea headings, etc.), the ShipMo7 COA-T service will round the requested parameters by itself as long as these are in the allowed bounds.

Table 6: SHIPMO_REQUEST message structure

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
Header	**JMSType**	**"SHIPMO_REQUEST"**	**String**	
Properties	**RequestID**	**Unique ID of the request**	**String**	
	APP_NAME	**Name of the sending application**	**String**	**Which application sent data (in our example, "SHIPMO-REQUESTER")**
Body *(JSON text string)*	**WaveHeight**		**Float**	
	WavePeriod		**Float**	
	BoatSpeed		**Float**	
	SeaHeading		**Float**	
	ShipType		**String**	**"halifax" or "mcdv"**

ShipMo7 Answer:

In the answer, the ShipMo7 service includes the parameters actually requested to the DB rather than the ones coming from the SHIPMO_REQUEST, for added transparency.

Table 7: SHIPMO_RESPONSE message structure

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
Header	**JMSType**	**"SHIPMO_RESPONSE"**	**String**	
Properties	**RequestID**	**Unique ID of the request**	**String**	

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
	APP_NAME	Name of the sending application	String	Which application sent data (in our example, "SHIPMO-SERVICE")
Body *(JSON text string)*	Resistance	Added Resistance in irregular waves	Double	
	KeelEmergence	Number of times the keel emerges from water	Double	Per hour
	DeckWetness	Number of times water rushes on the deck	Double	Per hour
	Mii	Number of times an operator has to brace himself	Double	Per minute
	WaveHeight		Float	Adapted from the request
	WavePeriod		Float	Adapted from the request
	BoatSpeed		Float	Adapted from the request
	SeaHeading		Float	Adapted from the request
	ShipType		String	Comes from the request, "halifax" or "mcdv"

ShortestPath Module

DepthRange Requests:

Given a longitude and latitude, it returns the depth range of this point.

Table 6: SHORTESTPATH_REQUEST message structure

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
Header	**JMSType**	**"DEPTH_RANGE_REQUEST"**	**String**	
Properties	**RequestID**	**Unique ID of the request**	**String**	
	APP_NAME	**Name of the sending application**	**String**	**Which application sent data (in our example, "SHORTESTPATH_REQUESTER")**
Body *(JSON text string)*	**lng**		**double**	**longitude**
	lat		**double**	**latitude**

ShortestPath Requests:

Even though mesh grid of the earth has a certain resolution (0.005 recommended) the ShortestPath COA-T service will round the requested latitude and longitude by itself as long as they are valid.

Table 6: SHORTESTPATH_REQUEST message structure

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
Header	**JMSType**	**"SHORTESTPATH_REQUEST"**	**String**	
Properties	**RequestID**	**Unique ID of the request**	**String**	
	APP_NAME	**Name of the sending application**	**String**	**Which application sent data (in our example, "SHORTESTPATH_REQUESTER")**
Body *(JSON text string)*	**start**		**double[]**	**[longitude, latitude]**
	end		**double[]**	**[longitude, latitude]**
	speed		**double**	Not currently in use
	decimal		**integer**	Number of decimal places
	Precision		**integer**	**1 or 5**
	maxDraft		**double**	Draft of the ship

ShortestPath Answer:

In the answer, the list of waypoints representing the path.

Table 7: SHORTESTPATH_RESPONSE message structure

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
Header	**JMSType**	**"SHORTESTPATH_RESPONSE"**	**String**	
Properties	**RequestID**	**Unique ID of the request**	**String**	
	APP_NAME	**Name of the sending application**	**String**	**Which application sent data (in our example, "SHORTESTPATH")**
	CLIENT_ID	**Unique ID of the application**	**String**	
	Possible	**Boolean information if the task is possible**	**Boolean**	
Body *(JSON text string)*	**finalPath**	**List of Point object representing the path**	**Double**	

DepthRange Answer:

In the answer, the range of depths at the requested point.

Table 7: DEPTH_RANGE_RESPONSE message structure

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
Header	**JMSType**	**"DEPTH_RANGE_RESPONSE"**	**String**	
Properties	**RequestID**	**Unique ID of the request**	**String**	
	APP_NAME	**Name of the sending application**	**String**	**Which application sent data (in our example, "SHORTESTPATH")**

JMS Part	Name *(String)*	Value *(Object)*	Value Type	Comment
	CLIENT_ID	**Unique ID of the application**	**String**	
Body *(JSON text string)*	**range**	**Array of depth from a point.**	**double[]**	**Format : [min, max]**

Annex B: PostgreSQL DB Schema

Since COA-T will use PostGIS as a Geo-Referenced DB for some of its application, this means that a PostGreSQL DB will run in the background. To save complexity, the Scheduler Service and the ShipMo7 module will refer to two different tables depending on the situation (halifax or mcdv) in the PostGreSQL DB (named shipmodata).

Describing the data model in this section allows a DB administrator to know what to expect in terms of content and format when working with the particular tables created through this call-up.

In both these tables, rows will iterate over the following parameters which serve us primary keys:

- Wave Height (0.2-11.2 metres, 0.2 increments)
- Wave Period (1-25 seconds, 0.2 increments)
- Ship Speed (1-30, 1 increments)
- Sea Heading, Wave direction relative to the ship (0-180, 5 increments)

From these input parameters, some values were extracted out of ShipMo 7:

- Added resistance in irregular waves (kN)
- Keel Emergence (per hour)
- Deck Wetness, Water washing the deck (per hour)
- Motion Induced Interruption (per minute)

Finally, everything was entered in the database in the following format:

Table 4: PostGreSQL DB Schema

Variable Name	Data Type
waveheight	Numeric
waveperiod	Numeric
boatspeed	SmallInt
seaheading	SmallInt
resistance	Double
keelemergence	Double
mii	Double
deckwetness	Double

Annex C: Application.properties

In this Annex, we show the content of the file *application.properties*, located in the *COATscheduler/src/main/resources* and *COATshipmo7/shipmo7/src/main/resources* directories. Each of the parameters listed in this file can be modified using an environment variable or as a Java option in the execution command. For example:

```
java -jar  -Djsa.activemq.broker.url= tcp://127.0.0.1:61616
COATscheduler/target/COAT-Scheduler-0.0.1-SNAPSHOT.jar
```

will change the broker URL parameter. If you want to change the default value permanently, you have to edit the *application.properties* file in the resource directory and then, recompile using the *mvn clean package* or *mvn clean install* command.

The unit for the timeout parameters are in milliseconds. The available logging levels are: ERROR, WARN, INFO, DEBUG, TRACE. For more information, see https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-logging.html .

Scheduler:

The Scheduler resource file is located in *COATscheduler/src/main/resources* directory.

The *mahalanobis.threshold* parameter determines how many standard deviations the weather parameters of a set of steps can be distant from another set before cutting the moving task.

The *addSmallTempest* parameter is for testing the scheduler with wave data that are over the task wave height limit and thus forcing a reschedule. It also serves to test the optimizer with wave data.

```
# PostgreSQL database connection configurations
spring.datasource.url = jdbc:postgresql://127.0.0.1:5432/shipmodata
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.datasource.platform=postgresql

# Active MQ configurations
jsa.activemq.topic.requests=C2.REQUESTS
jsa.activemq.topic.scheduleresponse=C2.REQUESTS_DATA
jsa.activemq.topic.weather=C2.INTERNAL_DATA.WEATHER

jsa.activemq.timeout=0

scheduler.addSmallTempest=false

jsa.activemq.broker.url=tcp://127.0.0.1:61616
jsa.activemq.broker.username=admin
jsa.activemq.broker.password=admin
```

```
spring.activemq.pooled=true
spring.activemq.pool.max-connections=4
spring.jms.pub-sub-domain=true

properties.mahalanobis.threshold=3

logging.level.org=INFO
```

ShipMo7:
The ShipMo7 resource file is located in *COATshipmo7/shipmo7/src/main/resources* directory.

```
# PostgreSQL database connection configurations

spring.datasource.url = jdbc:postgresql://127.0.0.1:5432/shipmodata
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.datasource.platform=postgresql

# Active MQ configurations

jsa.activemq.topic.requests=C2.REQUESTS
jsa.activemq.topic.scheduleresponse=C2.REQUESTS_DATA

scheduler.addSmallTempest=false

jsa.activemq.broker.url=tcp://127.0.0.1:61616
jsa.activemq.broker.username=admin
jsa.activemq.broker.password=admin
spring.activemq.pooled=true
spring.activemq.pool.max-connections=4
spring.jms.pub-sub-domain=true

logging.level.org=INFO
```

ShortestPath:
The shortestPath resource file is located in COAT-ShortestPath/src/main/resources directory. The S57 charts used for the algorithm must be in in COAT-ShortestPath/src/main/resources/s57files directory.

The *properties.depth.calculator* parameter determines which depth service is to be used. The value in this parameter must be equal to the value in the ConditionalOnProperty annotation over the chosen implementation of the depth service.

```
# Active MQ configurations
jsa.activemq.topic.requests=C2.REQUESTS
jsa.activemq.topic.shortestpathresponse=C2.REQUESTS.DATA

jsa.activemq.timeout=0
```

```
jsa.activemq.broker.url=tcp://192.168.48.19:61616
jsa.activemq.broker.username=admin
jsa.activemq.broker.password=admin
spring.activemq.pooled=true
spring.activemq.pool.max-connections=4
spring.jms.pub-sub-domain=true

properties.depth.calculator=s57service
properties.s57directory.path=pathToTheS57Directory

logging.level.org=INFO
```

DOCUMENT CONTROL DATA

*Security markings for the title, authors, abstract and keywords must be entered when the document is sensitive

1. ORIGINATOR (Name and address of the organization preparing the document. A DRDC Centre sponsoring a contractor's report, or tasking agency, is entered in Section 8.) OODA Technologies 4710 St-Ambroise, suite 226 Montreal, (Quebec) H4C 2C7 Canada	2a. SECURITY MARKING (Overall security marking of the document including special supplemental markings if applicable.) CAN UNCLASSIFIED 2b. CONTROLLED GOODS NON-CONTROLLED GOODS DMC A

3. TITLE (The document title and sub-title as indicated on the title page.)

PLANNING TOOL ALGORITHM IMPLEMENTATION FOR THE COURSE OF ACTION TEST BED

4. AUTHORS (Last name, followed by initials – ranks, titles, etc., not to be used)

Marion-Ouellet, L. O.; Bossé ,É.-O.; Bossé; Mayrand, M.

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total pages, including Annexes, excluding DCD, covering and verso pages.)	6b. NO. OF REFS (Total references cited.)
May 2018	48	0

7. DOCUMENT CATEGORY (e.g., Scientific Report, Contract Report, Scientific Letter.)

Contract Report

8. SPONSORING CENTRE (The name and address of the department project office or laboratory sponsoring the research and development.)

DRDC – Atlantic Research Centre
Defence Research and Development Canada
9 Grove Street
P.O. Box 1012
Dartmouth, Nova Scotia B2Y 3Z7
Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.) 01db	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.) W7707-4501665176
10a. DRDC PUBLICATION NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.) DRDC-RDDC-2018-C183	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11a. FUTURE DISTRIBUTION WITHIN CANADA (Approval for further dissemination of the document. Security classification must also be considered.)

Public release

11b. FUTURE DISTRIBUTION OUTSIDE CANADA (Approval for further dissemination of the document. Security classification must also be considered.)

12. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Use semi-colon as a delimiter.)
Scheduling; Operational Planning; Optimization and Matematical Programming; Optimization
13. ABSTRACT/RÉSUMÉ (When available in the document, the French version of the abstract must be included here.)
N/A