COMP6203 Intelligent Agents
Lab Instructions

Jan Buermann
edited by Jessica Newman (jln1g19@soton.ac.uk)

2025

Contents

Lab #1

1.1 Introduction

Imagine you are running a maritime shipping company that transports oil. You have a fleet of tankers around the world and you are looking for customers who want their oil to be transported from one port to another. Currently, the possible opportunities to transport cargoes, i.e. the oil, are as presented in Figure 1.

The refinery at Fawley (near Southampton) is looking for the cheapest shipper to transport two cargoes: one to Dublin and one to Rotterdam. You and your main competitor both have a vessel in Southampton and you now need to decide what price you would offer the refinery for the transportation of either cargo.

One option is to calculate how much it will cost you to transport either cargo and offer that as your price. Mind that, if you offer to transport both and you will get the contract for both you will have to come back to Southampton once you have dropped of the first.

Another consideration is that you know that soon another company will be looking for a shipping company to transport cargo from Rotterdam to Felixstowe. Hence, if you would get the contract to transport the cargo from Fawley to Rotterdam and the contract to transport the cargo from Rotterdam to Felixstowe, you can directly continue with transporting cargoes without travelling empty from one port to another.

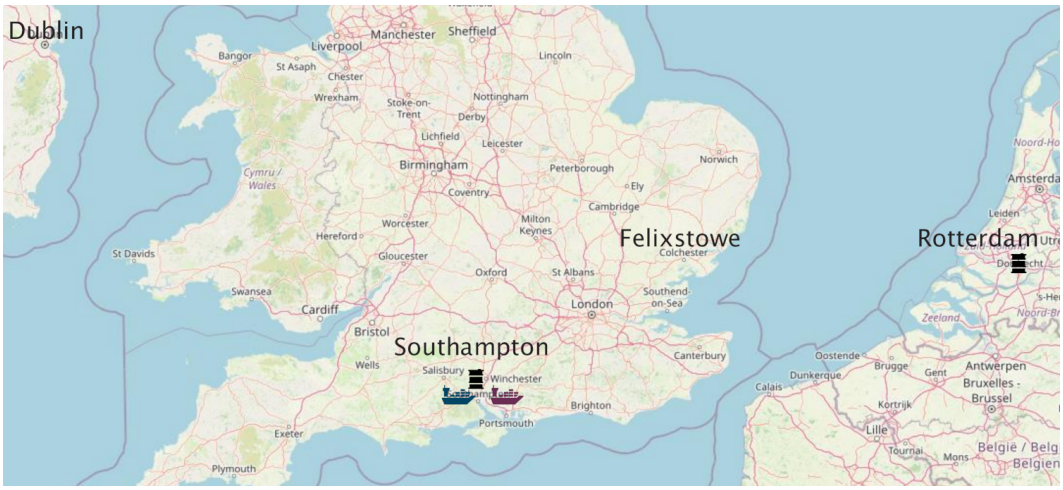

Figure 1: A maritime example.

This example highlights the general setting of the tramp trade maritime shipping which we will consider in the labs as well as in the coursework. The labs will guide you to understand how this setting is reflected in our simulator MABLE and develop a simple shipping company agent that will bid for cargo opportunities and schedules the

transportation of those cargoes they win the contracts for. In any setting including the competition at the end of the coursework, all shipping companies will have the option to bid for cargo opportunities at cargo auctions at regular intervals. The shipping companies will have to decide how much to bid at these reverse second-price auctions considering their already existing transport commitments, the requirement to transport all won contracts, some knowledge of future auctions and knowledge of the competitors' fleets.

1.2 Task 1: A first look at Mable

MABLE is an agent-based maritime cargo transportation simulator written in python which only needs a few steps to be set up. The simulator allows to specify a setting with ports, a cargo market with one homogeneous continuous cargo, e.g. oil, and shipping companies. Once such a setting is specified and run, the simulator generates random cargo transportation opportunities that shipping companies can bid for to transport and then have to be transported from one port to another. For the labs and the coursework the world with the ports and the cargo market are already specified and all you need to do is define the behaviour of a shipping company.

1.2.1 Good to know about MABLE

There are a few points about MABLE that it is good to be aware of.

Firstly, MABLE was developed following the object-oriented programming paradigm, meaning that everything is an object, e.g. vessels, ports, cargoes. If you are new to the concept or unsure about how this looks like in python there are courses on LinkedIn learning.

Secondly, MABLE is an event based simulator, meaning that time progresses in steps by events like a cargo auction happens, a vessel reaches a port or a vessel has transferred cargo. This will not affect your work as you will only need to specify in what order you vessels transport cargoes and MABLE will automatically translate that into events.

1.2.2 MABLE API

MABLE's API is accessible under MABLE API.

1.2.3 Setting up MABLE

MABLE requires at least Python version 3.11. For guidance on downloading and installing Python, go to `https://wiki.python.org/moin/BeginnersGuide/Download`.

MABLE is indexed on PyPi, so if you are running a version of Python >=3.11 you should be able to install MABLE using pip with the following command:

```
pip install mable
```

If this doesn't work you can also download MABLE manually:

- Download the simulator wheel `mable-0.0.13.post3-py3-none-any.whl`
- Install MABLE in your local python, e.g.

  ```
  pip install mable-0.0.13.post3-py3-none-any.whl
  ```

IMPORTANT: Some of the MABLE code interfacing with another package is now deprecated. If you are getting the error `Field.__init__() got an unexpected keyword argument 'default'`, try installing the following wheel instead: **Fixed MABLE wheel**

To run MABLE, you will need to download the resources file `mable_resources.zip` (alternate download link) and place this in the working directory of your python project.

1.3 Task 2: Running a Maritime Setting

To implement your own shipping company you will have to populate a subclass of `mable.cargo_bidding.TradingCompany`. We will start with a simple setup with an empty shipping company that just does the default behaviour.

1.3.1 Simple setup

Getting started with a running example is fairly straightforward with the provided example functions. We will be creating the following frame.

```
from mable.cargo_bidding import TradingCompany
from mable.examples import environment, fleets

class MyCompany(TradingCompany):
    pass

if __name__ == '__main__':
    specifications_builder = environment.
    ↪  get_specification_builder(environment_files_path=".")
    fleet = fleets.example_fleet_1()
    specifications_builder.add_company(MyCompany.Data(MyCompany, fleet,
    ↪  "My Shipping Corp Ltd."))
    sim = environment.generate_simulation(specifications_builder)
    sim.run()
```

The steps for that are the following.

1. Create a new python file.
2. Start with the imports.
   a) Import `TradingCompany` from `mable.cargo_bidding` to load the shipping company base class
   b) Import `environment` and `fleets` from `mable.examples`. These allow you to quickly set up a default trading environment and predefined fleets of vessels.
3. Create the shipping company class by creating a class with a name of your choosing ,e.g. `MyCompany`, that is a subclass of `TradingCompany`. We will leave the class definition empty, using the the default implementation of the `TradingCompany` class (The default implementation can be seen here - don't worry if you don't understand the functions defined in here, as we will be going over them in future labs!)
4. A simulation is created via adding details to a *specifications_builder*. `mable.examples.environment.get_specification_builder()` returns an instance of such a builder and takes care of further setup like linking the auxiliary files. The function has one parameter `environment_files_path` which allows the specification of the directory of the resources file (the `mable_resources.zip`). The default is the current directory where the script will be executed (".").
5. Generate a sample fleet with `mable.examples.fleets.example_fleet_1()`
6. Use the `specifications_builder` and the `add_company` function to add your company.
   - To define a company you need to specify the class, the fleet and the name of the company.
   - The function expects a `DataClass` object which is automatically provided via `MyCompany.Data`. Hence you can create the company specification via `MyCompany.Data(MyCompany, fleet, "My Shipping Corp Ltd.")`
7. This specifies a simulation which can be created for the *specifications_builder* using the `environment.generate_simulation` function which returns a simulation object. Hence,

   ```
   sim = environment.generate_simulation(specifications_builder)
   ```

   provides the simulation.
8. Finally, you can run the simulation by calling the simulation's `run` function, i.e. `sim.run()`.

1.3.2 Run the maritime Setting

Once you have created a simple setup as describe in Section 1.3.1 you can run the file to start the simulation. While the script is running it should print to the console events

that are happening. The run should finish with the log message `---Run Finished---`. The run will produce a file that contains information about the companies' operation and the outcomes of the cargo auctions, called `metrics_competition_<number>.json` where <*number*> is a random id. To get a quick overview of the income, MABLE comes with a little script to print this information to the console. Simply call

```
mable overview metrics_competition_<number>.json
```

which should produce an output like the following.

```
Company My Shipping Corp Ltd.
+---------+---------+
|   Name  |   Value |
+---------+---------+
|   Cost  |    1000 |
| Penalty |       0 |
| Revenue |    1100 |
+---------+---------+
|  Income |     100 |
+---------+---------+
```

1.3.3 What is the code doing?

Take a look through the MABLE documentation here. See if you can find some of the classes we used in the example code, and consider what their function might be.

A python *.whl* file is actually a ZIP-format archive with a specially formatted file name. This means you can see the source code for MABLE by unzipping the `mable-0.0.13.post3-py3-none-any.whl` file available here, or by looking at the source files in the MABLE github repository. If you are unsure of how a certain class or method works, it can be very helpful to look at the source code!

Lab #2

2.1 The World

The world or environment in MABLE is a shipping network of connected ports. Each port has a name and a location consisting out of a latitude and longitude. (While you won't have to interact with ports directly, ports are represented by `class LatLongPort` with latitude and longitude in decimal degrees.)

2.1.1 Transportation Opportunities (TimeWindowTrade)

Cargo transportation opportunities the companies can bid for are represented by `class TimeWindowTrade` (see Figure 2). Each of these has specifies a port from which the cargo has to be picked up, `origin_port`, and a port at which the cargo has to be dropped off, `destination_port`. Moreover, the trade specifies what `amount` of which type (`cargo_type`) has to be transported.

There will be only one `cargo_type` which is `"Oil"`. However, you can still use the property whenever you need to pass the type to a different function.

Additionally, the trade may specify up to four time points which indicate restrictions on the time when the cargo can be picked up and dropped off.

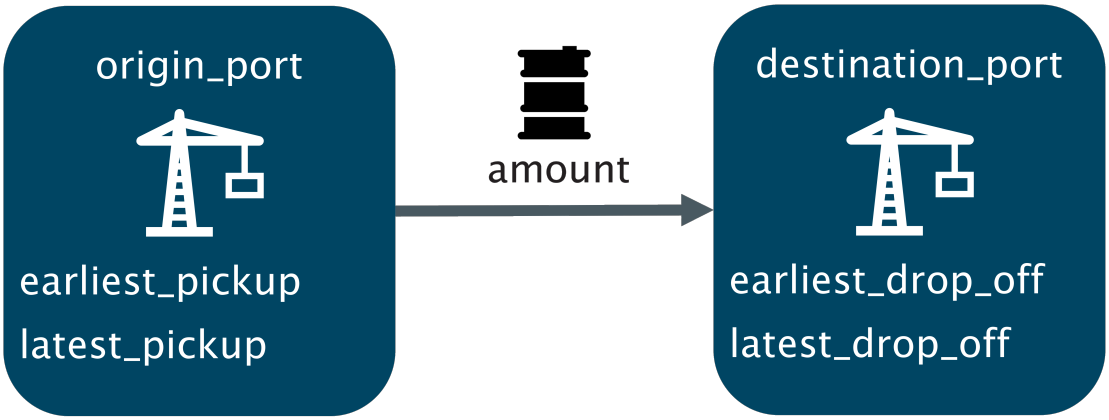

Figure 2: Illustration of a time window trade.

- `earliest_pickup` When the vessel arrives earlier it will have to wait.

- `latest_pickup` When the vessel arrives later it will not be able to pick up the cargo, i.e. fulfil the job.
- `earliest_drop_off` When the vessel arrives earlier it will have to wait.
- `latest_drop_off` When the vessel arrives later it will not be able to drop off the cargo, i.e. fulfil the job.

If a time is not set, i.e. no restrictions on arrival, this is indicated by None for the respective time.

You can ignore all other instance variables of `class TimeWindowTrade`: `probability` and `status`. All trades will happen.

2.1.2 Vessels (VesselWithEngine)

Cargoes are transported using vessels (ships) which have a cargo hold to store the cargo and and engine that burns fuel. Each company has a number of vessels whose cargo hold size and fuel consumption might be different. The cargo hold size restricts the amount the vessel can transport. The fuel consumption depends on what the vessel is doing but even when the vessel is idling the vessel consumes fuel. There are five different states or operations of what the vessel could be doing (see Figure 3), each have their own rate of consumption which is an amount of fuel per time:

- Idling, i.e. doing none of the other operations.
- Loading, i.e. loading cargo within a port.
- Unloading, i.e. unloading cargo within a port.
- Laden, i.e. moving between ports with cargo in the hold.
- Ballast, i.e. moving between ports with no cargo in the hold.

Figure 3: Vessel statuses.

The class you will have to interact with in MABLE is the `class VesselWithEngine` which models a cargo hold and an engine that consumes fuel. VesselWithEngine (including via its ancestors[1]) has a lot of convenience methods to calculate capacity, loading rate, times for loading, consumptions and costs.

2.2 The Market

Periodically an auction will be run at which several transportation opportunities (see Section 2.1.1) will be on offer. All companies can bid to transport any or all of the available trades by offering a price per trade they would like to get paid for the transportation of the particular trade. The market will collect all bids and for each trade will run a second cost reverse auction to determine the winner and the payment to the winning company. Cargoes that have been awarded to a company have to be transported by that company. Otherwise, the company will be fined.

The market will also inform companies of upcoming auctions beforehand. Just before an auction - the current auction - the market will inform all companies what trades will be available at the auction that will follow in the next period after the current auction.

2.3 Running a Company

For the companies to interact with the market there is an interface of three functions: `pre_inform`, `inform` and `receive`. The body of these functions is the main way you will provide behaviour to your agent.

These are called in three different stages during the simulation.

First, whenever an auction is due to happen at a future time, the market will call the `pre_inform` function of your company with two parameters: the trades which are to go up for auction, and the time at which the auction will occur. You can add some behaviour to your agent whenever this occurs.

Following this, the market will hold an auction for the trades in the current auction round. The market will use the `inform` function to again tell the companies which trades are auctioned off. When this function is called, the company must return a list of bids for each trade going up for auction. The expectation is that the companies return a list of all bids they want to place as a list of `class Bid`. You can change the body of this function to determine how much you wish to bid for each trade up for auction.

The market will use all bids from all companies to determine the winners for all trades in the current auction round. Each auctioned trade will go to the company who bid the lowest for that trade. On completion of the trade, the company is paid the amount of the second-lowest bid for that trade. Once an auction is complete, the market will call `receive` to inform every company of the trades they won in the auction. (The call will also contain the information about all auction outcomes which may be of interest to opponent modelling. This will be discussed in Lab #3.) It is useful to use the body

[1] `WorldVessel`, `SimpleVessel` and `Vessel`

of this function to determine how you are going to schedule any new trades you won at auction: which vessels you will assign them to, and it what order they should be completed.

2.4 Schedules

A schedule is the plan of operation for a single vessel which captures the order of when the cargo of a trade gets picked up and when the cargo of the trades gets dropped off. You can think of a schedule as a simple sequence of pick-up and drop-off events. For example a vessel that is supposed to transport two trades, $trade_1$ and $trade_2$, might transport both of them in order or pick up both of them first and then drop them off. This results in six possible combinations.

- [Pick up $trade_1$, Drop off $trade_1$, Pick up $trade_2$, Drop off $trade_2$]
- [Pick up $trade_2$, Drop off $trade_2$, Pick up $trade_1$, Drop off $trade_1$]
- [Pick up $trade_1$, Pick up $trade_2$, Drop off $trade_1$, Drop off $trade_2$]
- [Pick up $trade_1$, Pick up $trade_2$, Drop off $trade_2$, Drop off $trade_1$]
- [Pick up $trade_2$, Pick up $trade_1$, Drop off $trade_1$, Drop off $trade_2$]
- [Pick up $trade_2$, Pick up $trade_1$, Drop off $trade_2$, Drop off $trade_1$]

The `class Schedule` provides functionality to plan schedules, including adding to and verifying schedules.

2.5 Exercise 1: Creating a Schedule

Last week we created a subclass of `class TradingCompany`. This week we will extend it by writing how our company schedules the transportation of trades and later how to bid for trades. Starting with the scheduling of trades, the default implementation of `inform` calls the `propose_schedules` function of the company object and expects the function to return a list of trades that can be transported, the proposed schedules to do this and the estimated costs for each cargo. In your first task you will override this function `def propose_schedules`.

Before we start, the `propose_schedules` specifies the return value, consisting of the list of trades, the proposed schedules and the estimated costs, to be wrapped in an instance of the `class ScheduleProposal` data class. So you need to import the class.

Getting Ready

- We will start with your code from last week.
- Add the following import.

```
from mable.transport_operation import ScheduleProposal
```

2.5.1 Exercise 1.1: A Very Simple Scheduling

Let's add a simple implementation of `def propose_schedules`. By default, this function is called when we obtain some new trades from an auction, and we wish to assign these trades to the schedule of one of our vessels. It is also called before an auction to see how we would schedule a trade should we win.

For our simple implementation, for each new trade we will just append this to end of one of our vessel's existing schedules. We want to ensure our vessel can actually carry out this trade in the required time window, however (remember from Section 2.1.1, every trade has a latest pickup and dropoff time!) Luckily, MABLE has some helper functions for this. The idea will be to iterate over the list of trades and the list of vessels and check if for every trade-vessel combination the vessel can transport the cargo after finishing their current jobs.

Hence, we will do the following.

- Add `def propose_schedules (self, trades)` to your company class.
- Add code to `propose_schedules`.
- For now we will ignore the costs. So, you can just return {} for the `costs` parameter of `ScheduleProposal`.

Your starting point should look like this.

```
from mable.cargo_bidding import TradingCompany
from mable.examples import environment, fleets
from mable.transport_operation import ScheduleProposal

class MyCompany(TradingCompany):

    def propose_schedules(self, trades):
        schedules = {}
        costs = {}
        scheduled_trades = []
        # Your Code
        return ScheduleProposal(schedules, scheduled_trades, costs)

if __name__ == '__main__':
    specifications_builder = environment.␣
    ↪  get_specification_builder(environment_files_path=".")
    fleet = fleets.example_fleet_1()
    specifications_builder.add_company(MyCompany.Data(MyCompany, fleet,
    ↪  "My Shipping Corp Ltd."))
    sim = environment.generate_simulation(specifications_builder)
    sim.run()
```

You will notice we need to give our return `ScheduleProposal` object three values. These are:

- `schedules`: this is a dictionary where the keys are of type `Vessel` and the values are of type `Schedule`. This tells the simulator which schedules we will be assigning to each ship.
- `scheduled_trades`: this is a list of type Trade. This tells the simulator which trades we have scheduled.
- `costs`: this is a dictionary with keys of type `Trade` and values of type `cost`. By default, if `propose_schedules` is called during an auction in the `def inform` function, this will be used to determine how much you will bid for that trade.

Under the comment `#Your Code` we want to implement the following:

- For each trade in `trades` ...
  - For each vessel in your fleet...
    - Get this vessels current schedule
    - Create a copy of this schedule
    - Add the trade to the end of this schedule
    - Check if this trade can be carried out in time according to this schedule
    - if not, ignore and move onto the next vessel
    - if so, add the key value pair (vessel, new schedule) to the `schedules` dict and add the trade to the `scheduled_trades` list

Make sure you only assign each trade to a schedule *once* - you don't want the same trade in multiple ship's schedules!

When you get a vessels current schedule, you may have already added a trade to it in the current call of `propose_schedules`. Since during the loop you will be adding updated schedules to the `schedules` dict, when you pull a vessel's current schedule you should first try and find an updated version in the `schedules` dict. If the vessel doesn't appear in here, the vessels schedule has not been updated so far in the current call of `propose_schedules`, so you are safe to pull the schedule from the vessel object directly using the `.schedule` property

To complete the code the following functions already provided by MABLE will help you.

- `def fleet` Your company class has a `fleet` property which gives you a list of data on your company's fleet of vessels.

- While in the class `MyCompany`, you can use the code `self._fleet` to return a list of the vessel objects in your company.
- `def schedule` Every vessel has a `schedule` property to get a copy of its current schedule. For example, the code `self._fleet[0].schedule` will return the schedule object for the first vessel in your fleet. Do not use `Schedule` directly to create new schedules. They will not be set up correctly and you will encounter errors.
- `def add_transportation` `add_transportation` allows you to add a trade to a schedule. If the function is called specifying only the trade, the function will append the trade to the end of the schedule.
- `def verify_schedule` Once you have constructed a schedule, you can use the function `verify_schedule` to verify that the schedule can be fulfilled. Fulfilment, is based on two factors, the ability to deliver on time and the condition that the cargo hold size is not violated.
  - All cargoes are delivered on time if all cargoes are picked up and dropped off within the time windows specified by the trades. This can also be tested individually by using the `def verify_schedule_time` function.
  - The cargo hold size is not violated if the vessel is never overloaded, i.e. the cargo is not exceeding the capacity of the cargo hold, never underloaded, i.e. trying to unload cargo when the vessel is empty. This can also be tested individually by using the `def verify_schedule_cargo` function.
- `def copy` For convenience the `Schedule` has a `copy` function which gives you a deep copy of the schedule, i.e. you can change the copy without affecting the old schedule. Hence, you can create different alternatives without make new variants of the schedule without messing up the original schedule. You might not need this function right now but remember it for later!

`def inform` will simply take the list of scheduled trades and for now bid zero on them.

The `class Schedule` defines a range of other functions to interact with the schedule which are used by the simulation: `def next`, `def pop`, `def get`, `def __getitem__` (the dunder to allow the syntactic sugar access via square brackets, i.e. `some_schedule[i]`). You should **not** use these! `Schedule` does a lot of 'under-the-hood' work to establish, maintain and verify schedules which could be messed up when you do not use these function in the right way.

2.5.1.1 Solution

Here is a sample solution for the above task. Please make sure you understand which classes each part of the functionality is coming from, as this will be useful when you design your own agent!

```
class MyCompany(TradingCompany):

    def propose_schedules(self, trades):
        schedules = {}
        scheduled_trades = []
        i = 0
        while i < len(trades):
            current_trade = trades[i]
            is_assigned = False
            j = 0
            while j < len(self._fleet) and not is_assigned:
                current_vessel = self._fleet[j]
                current_vessel_schedule = schedules.get(current_vessel,
                    current_vessel.schedule)
                new_schedule = current_vessel_schedule.copy()
                new_schedule.add_transportation(current_trade)
                if new_schedule.verify_schedule():
                    schedules[current_vessel] = new_schedule
                    scheduled_trades.append(current_trade)
                    is_assigned = True
                j += 1
            i += 1
        return ScheduleProposal(schedules, scheduled_trades, {})
```

2.5.2 Exercise 1.2: Slightly Better Scheduling

Sometimes two cargo pick up locations may be close to each other and it would make more sense to pick up both cargoes before dropping them off (assuming the cargo hold is big enough) instead of simply appending full jobs one after another.

The `def add_transportation` we used to add trades to a schedule actually has more sophisticated functionality. Instead of just appending a trade to the end of a schedule, we can decide when we want to pick up the cargo for that trade, and when we want to drop off the cargo for that trade.

The `Schedule` class has a function `get_insertion_points` which allows you to get all valid locations that can be used for the `location_pick_up` and `location_drop_off` of the `add_transportation` function. You could use this to determine if instead of just appending a trade to the end of a schedule you could do the pick up and drop off earlier similar to some of the combinations in Section 2.4. We could adapt our last exercise to calculate exactly where it is optimal to insert our new trade into our current schedule.

Hence, we will do the following.

- Take your code from Exercise 1.1 (Section 2.5.1).
- Modify `propose_schedules` to have the following functionality:
- For each new trade:
  - Select a vessel's current schedule.
  - Get all the possible insertion (pick up and drop off) points for the trade into this schedule.
  - For each possible combination of pick up and drop off points:
    - Create a copy of the schedule with the new trade inserted at these points using `add_transportation`
    - Verify whether this schedule can be fulfilled.
    - Estimate the time it would take to carry out this new schedule.
  - (Of the schedules than can be fulfilled) Take the schedule with the insertion points that had the lowest time estimate, and add this to the `schedules` dict as well as the trade to the `scheduled_trades` list as in the previous exercise.
  - If none of the insertion points gave a schedule that can be fulfilled, try the trade on a different vessel's current schedule.

You may find the (`completion_time`) function in the `Schedule` class helpful, which returns an estimate of the time taken to complete a schedule based on some precomputed distances in the resources.zip file.

2.5.2.1 Solution

```
class MyCompany(TradingCompany):

    def propose_schedules(self, trades):
        schedules = {}
        scheduled_trades = []
        i = 0
        while i < len(trades):
            current_trade = trades[i]
            is_assigned = False
            j = 0
            while j < len(self._fleet) and not is_assigned:
                current_vessel = self._fleet[j]
                current_vessel_schedule = schedules.get(current_vessel,
                    current_vessel.schedule)
                new_schedule = current_vessel_schedule.copy()
```

```
                insertion_points = new_schedule.get_insertion_points()
                shortest_schedule = None
                for k in range(len(insertion_points)):
                    idx_pick_up = insertion_points[k]
                    insertion_point_after_idx_k = insertion_points[k:]
                    for m in range(len(insertion_point_after_idx_k)):
                        idx_drop_off = insertion_point_after_idx_k[m]
                        new_schedule_test = new_schedule.copy()
                        new_schedule_test.
                            add_transportation(current_trade,
                            idx_pick_up, idx_drop_off)
                        if (shortest_schedule is None
                                or new_schedule_test.completion_time() <
                                    shortest_schedule.
                                    completion_time()):
                            if new_schedule_test.verify_schedule():
                                shortest_schedule = new_schedule_test
                if shortest_schedule is not None:
                    schedules[current_vessel] = shortest_schedule
                    scheduled_trades.append(current_trade)
                    is_assigned = True
                j += 1
            i += 1
        return ScheduleProposal(schedules, scheduled_trades, {})
```

2.6 Exercise 2: Creating a Bid

So far we have left the `costs` field in `ScheduleProposal` empty. The `TradingCompany` function `def inform` is called whenever an auction is being held, and returns a list of bids for trades available in that auction. The default behaviour is to call `propose_schedules`, and use the `costs` field to determine how much to bid.

Next we will extend our `propose_schedules` to calculate transportation costs and use that for the bids, by returning this in the `costs` field. Your function should now be able to schedule new trades, so once you have found a working schedule you can check how long it would take to:

- Load the cargo,
- Travel from the origin port to the destination port,
- Unload the cargo.

If we added this new trade to the schedule. We can use these times to calculate the fuel consumption for each of these costs. This will allow us to decide how much this trade is worth to us, and so how much we should bid.

To get started:

- Take either your code from Exercise 1.1 (Section 2.5.1) or from Exercise 1.2 (Section 2.5.1).
- Modify `propose_schedules` to also calculate how much the transportation might cost.
  - You can focus on the loading, unloading and transportation costs for now.
  - You should build up a dictionary with (key,value) pairs of trades as the keys and floats as the values. This should store for each trade the calculated cost of this trade. This should be returned in the costs field of the `ScheduleProposal`.

```
def propose_schedules(self, trades):
    schedules = {}
    scheduled_trades = []
    costs = {}
    i = 0
    while i < len(trades):
        current_trade = trades[i]
        is_assigned = False
        j = 0
        while j < len(self._fleet) and not is_assigned:
            current_vessel = self._fleet[j]
            current_vessel_schedule = schedules.get(current_vessel,
            ↪  current_vessel.schedule)
            new_schedule = current_vessel_schedule.copy()
            new_schedule.add_transportation(current_trade)
            if new_schedule.verify_schedule():
                # TODO calculate costs
                costs[current_trade] = 0  # TODO Add cost
                schedules[current_vessel] = new_schedule
                scheduled_trades.append(current_trade)
                is_assigned = True
            j += 1
        i += 1
    return ScheduleProposal(schedules, scheduled_trades, costs)
```

Again, to complete the code you can use the functions already provided by MABLE.

- To determine times and fuel consumptions there are a lot of convenience functions in the vessel classes, as mentioned in Section 2.1.2.
- `def get_network_distance` Your company class has a property `def headquarters` which allows you to access its `class CompanyHeadquarters`. This class provides the `get_network_distance` to calculate the distance between two ports.

- Vessel objects have a `get_travel_time` method which can return the time it takes to travel a given distance. Similarly, there is a `get_laden_consumption` method for determining how much fuel it takes to travel a certain distance. Looking through the documentation, you can find similar methods to determine the fuel it takes to unload and load a certain amount of cargo.

The cost of fuel is set to 1 per unit of fuel. Therefore you can just take the amount of fuel consumed to be the cost.

`def inform` will simply take the list of scheduled trades and the costs and will bid for those trades using the costs as the bid value. Its implementation looks like this.

```
            def inform(self, trades, *args, **kwargs):
                proposed_scheduling =
                ↪  self.propose_schedules(trades)
                scheduled_trades =
                ↪  proposed_scheduling.scheduled_trades
                trades_and_costs = [
                    (x, proposed_scheduling.costs[x]) if x
                    ↪  in proposed_scheduling.costs
                    else 0
                    for x in scheduled_trades]
                bids = [Bid(amount=cost, trade=one_trade)
                ↪  for one_trade, cost in trades_and_costs]
                return bids
```

The `def receive` function will simply apply `def propose_schedules` to the won trades and uses these schedules. For your own agent you might want to have a different procedure to schedule these cargoes.

2.6.1 Solution

```
class MyCompany(TradingCompany):

    def propose_schedules(self, trades):
        schedules = {}
        costs = {}
        scheduled_trades = []
        i = 0
        while i < len(trades):
```

```
            current_trade = trades[i]
            is_assigned = False
            j = 0
            while j < len(self._fleet) and not is_assigned:
                current_vessel = self.fleet[j]
                current_vessel_schedule = schedules.get(current_vessel,
                ↪  current_vessel.schedule)
                new_schedule = current_vessel_schedule.copy()
                new_schedule.add_transportation(current_trade)
                if new_schedule.verify_schedule():
                    loading_time = current_vessel.
                    ↪  get_loading_time(current_trade.cargo_type,
                    ↪  current_trade.amount)
                    loading_costs = current_vessel.
                    ↪  get_loading_consumption(loading_time)
                    unloading_costs = current_vessel.
                    ↪  get_unloading_consumption(loading_time)
                    travel_distance =
                    ↪  self.headquarters.get_network_distance(
                        current_trade.origin_port,
                        ↪  current_trade.destination_port)
                    travel_time =
                    ↪  current_vessel.get_travel_time(travel_distance)
                    travel_cost = current_vessel.
                    ↪  get_laden_consumption(travel_time,
                    ↪  current_vessel.speed)
                    costs[current_trade] = loading_costs +
                    ↪  unloading_costs + travel_cost
                    schedules[current_vessel] = new_schedule
                    scheduled_trades.append(current_trade)
                    is_assigned = True
                j += 1
            i += 1
        return ScheduleProposal(schedules, scheduled_trades, costs)
```

2.7 Having trouble?

If you are stuck or want to check your solution, sample code for the above exercises can be found here. Make sure you try the exercises first!

2.8 Running Tournaments

You may have noticed the default simulation specification does not have many trades available. You can adjust how many auctions run and how many cargoes are auctioned off to test different settings with different numbers of vessels and companies. The methods to do this can be found in `def get_specification_builder` and its parameters.