MoSIS Assignment 6

Team

Name	: Gauthier Le Compte
Student ID	: s0172093
e-mail	: gauthier.lecompte@student.uantwerpen.be

Name	: Nick Wils
Student ID	: s0162945
e-mail	: nick.wils@student.uantwerpen.be

Time spent on this assignment (in hours): 125

Note

The plots and output for task 5, 6, 7 and 8 can be found in the **model** folder under **plots**. The images included in this report, such as the drawing of our model, can be found in the **resources** map.

The code of this project can also be found at: https://github.com/NickWils98/MoSIS6.git

MoSIS Assignment 6 **1**
Team 1
Note 1
Task 1: Individual Components 3
Messages/Events 3
Vessel 3
Port Events 3
Atomic Devs 3
Generator 3
Anchorpoint 4
Sea 4
Waterway 5
Canal 6
Confluence 6
Lock 7
Dock 8
Control Tower 8
Task 2: Drawing of Model 10
Task 3: Coupled DEVS Model 11
Task 4: Experimentation 12
Task 5: 100 vessel generation 13
Task 6: 1 week simulation 15
Seed 42 15
Seed 69 16
Task 7: Alter constant values 17
Change lock interval 17
Change probabilities 17
Task 8: Deterministic Model 18

Devs Assignment

Task 1: Individual Components

An important choice we made is the units in our implementation. We chose km/h for all speeds. So everything was transformed to kilometres and hours. Furthermore for the time in our simulation we chose that 1 count is 1 hour. So 4u30 is 4.5 in hour simulation.

Messages/Events

Vessel

messages_events/vessel.py
To create the vessels we used a factory. The factory has a create function that will return a vessel based on the probabilities given.

We have 1 superclass Vessel that keeps track of the randomly generated name, creation_time, id, destination and the time it entered the port. For each of the different types of vessels, we created a subclass that inherits from the superclass Vessel. These subclasses contain all the info specific to that type of vessel, such as the average speed.

Port Events

messages_events/port_events.py
For the 3 different port events, “PortEntryRequest, PortEntryPermission, PortDepartureRequest”, we created the **port_events.py** file that contains 1 class for each event. Depending on what the message is, different variables are kept. For the *portEntryRequest* we keep the vessel_id and current_time, for the *portEntryPermission* that sends the control tower to the anchor point we again keep the vessel_id, but also the destination (quay_id) and current_time. Finally for the *portDepartureRequest* we keep track of the vessel_id and quay_id.

Atomic Devs

All the AtomicDEVS models we are about to discuss have a state and “normal”/AtomicDEVS class. The state class defines the state of the AtomicDEVS class as a structured object of the normal class

Generator

atmoic_devs/generator.py
The GeneratorState is in charge of generating the vessels. To do this it keeps track of how many vessels are created each hour, the current time, remaining time until the new vessel needs to be created. The Generator class itself keeps a factory as well to create the vessels. There is 1 out port to send the vessels to the anchorpoint and another one to send the total vessels generated to the collector. Since the Generator doesn’t receive any inputs, it also doesn’t have an extTransition function. The intTransition function on the other hand is called to after the remaining time has expired, to create the next vessel and start a new timer.

The timeAdvance function returns either infinity when we are done producing vessels, so it remains idle, or it returns the remaining time to the next vessel. In the outputFnc we create a vessel and put it in the output port such that it can be sent to the anchorpoint.

Anchorpoint

atmoic_devs/anchorpoint.py
The AnchorpointState keeps several queues:

- A queue with vessels that just came in
- A queue with vessels that requested to enter the port
- A queue with vessels that are about to leave

And it also holds a list of requests to send.

The AnchorPoint AtomicDEVS class has 1 input port in which the vessels arrive and one output port for sending vessels on the first waterway. It also has input and output ports for events such that communication with the control tower can be established. Finally, there is another output port which sends data, for statistics and performance monitoring, to the collector state.

Because the output of the generator is connected with the input of the anchorpoint, whenever a generated vessel is sent through the output port of the generator, the extTransition function is triggered. A vessel is then added to the waiting queue. Another possibility is that a message is received from the in_event port. This message comes from the control tower and identifies which vessels are allowed to enter and to which quay/dock they must travel.

The intTransition function generates a portEntryRequest to ask the control tower to which quay/dock they must sail. This request is added to a list such that these can be processed later on. The corresponding vessel is added to the queue with vessels that requested to enter the ports.

The timeAdvance function returns 0 if there are still vessels in the leaving or waiting list because we don't wait if there is a message or vessel waiting to send. If there is no vessel or request in the queue, then the state will remain idle and infinity is returned.

Sea

atmoic_devs/sea.py
The sea only has one in_port and a list which keeps track of which vessels have sailed into the sea. The only outport is used to send the total vessels who left via the sea to the collector. The input port of the sea is connected to the output port of the waterway which travels from anchorpoint to the sea. The only functionality of the class is to keep the left vessels in a list and report to the collector.

Waterway

atmoic_devs/waterway.py
In waterways we assume that a vessel will travel in total at the average speed.

Waterways are AtomicDEVS which connect different other classes. The state of the Waterway holds its distance. Since vessels can travel in both ways, a lot of the code is duplicated for both directions. We have an ingoing and outgoing dictionary where the key is the vessel and the value is the remaining time before arrival of that specific vessel. It speaks for itself that the ingoing dictionary holds the vessels going in one way, and the outgoing holds the other direction.

The waterway has input and output ports for one way as well as input and output ports for the other way. So in total it has 2 in ports and 2 out ports.

When a vessel is in one of the ports, the extTransition is fired. This will first update all the remaining times of the vessels which are already on the waterway. When doing this update, it may be possible that the remaining time of a vessel will be equal to zero. When this is the case, we set the value of the remaining time in the dictionary equal to zero. Such that we can take care of this in the intTransition function without delay. When we have lowered the remaining times, we add the vessel waiting in the input to the ingoing or output dictionary, and calculate its remaining time by dividing the distance of the waterway by the average velocity of the vessel.

The timeAdvance function is going to choose the correct remaining time. The correct remaining time in this case is the vessel with the shortest time. So we loop over both the ingoing and outgoing dictionaries and find the shortest time between the vessels. This is important because in the intTransition we are going to use this whether a vessel has arrived or not. When a vessel is ready to leave, which is specified in the inTransition, we will set the remaining time to 0. This way the vessel can leave without delay.

In the intTransition we will let vessels whose timer have expired leave:
1. We update all the remaining times for the vessels by subtracting the remaining time we waited. If the vessel’s remaining time is less than 0, it means the vessel has arrived and we add it to ingoing_leaving list.
2. We loop over the ingoing_leaving list and delete the arrived vessels from the dictionary.

We do the exact same process for the vessels in the ingoing dictionary as we do in the outgoing dictionary, so this process is identical.

Then finally for the outputFnc, we output all the vessels who left the waterway in the right direction and reset the outgoing and ingoing leaving lists.

Canal

atmoic_devs/canal.py
A canal is basically a waterway where vessels are not allowed to pass each other. Thus we have to keep track of which vessel is in front of it. The velocity of the vessel becomes the minimum of your velocity and the velocity of the vessel in front.

Since vessels are not able to pass each other in a canal, we had to use a queue of lists instead of dictionaries. For example for the ingoing queue it had the following structure:

- Ingoing [[vessel1: 50] , [vessel2: 100], [vessel3: 150], ….]

So the lists have the same structure as in the waterway, but we keep them in a queue now because we have to maintain a FIFO system.

The timeAdvance, outputFnc and intTransition all follow exactly the same principle as the waterway, but then with a queue instead of a dictionary of course, hence we will not discuss this again.

The extTransition on the other hand is a bit different because we have some extra computation work we have to do. First we update the remaining times of the ingoing and outgoing vessels just like we did in the waterway. But when looping over all the inputs, we check If it is the first vessel, because then we do not need to take into account the velocity of the vessel in front. If it's not the first vessel, we do need to take into account the velocity of the vessel in front because we have to take the min of the two vessel velocities.

When we have a slow vessel and a fast vessel in the same canal, with the slow vessel in front. The fast vessel reduces its speed to match the slow vessel. When that slow vessel leaves the canal, the faster vessel can technically speed up. This is not something we implemented. The faster vessel will keep the slower speed until it leaves the canal.

Confluence

atmoic_devs/confluencel.py
atmoic_devs/confluence_port.py
The state of the confluence has a queue and a map_port. The queue ensures FIFO for letting vessels leave. The map is a list that has a list for each split. Each of those lists has all the destinations that are the shortest using this split. The initialization process of the confluence goes as follows:

Confluence_KS = self.addSubModel(Confluence([["K"], [1, 2, 3, 4, 5, 6, 7, 8], ["S"],], 3))

The first part [["K"], [1, 2, 3, 4, 5, 6, 7, 8], ["S"],] is called the map_port and the last number 3 is the amount of in and outports the confluence has. All of the confluences have 3 in and outports, except for the one in Lock C, this one has 5.

To explain the map port we will again use the example given above. Since we have 3 outputs, we also have to have 3 maps. The first list indicates, that the fastest way to reach "K", is via the output port 0, the fastest way to reach dock 1, 2, 3, 4, 5, 6, 7, or 8 is via output port 1, and finally the fastest way to reach Sea "S" is via output port 2.

The timeAdvance function returns either 0 or infinity, depending on whether it has still elements in the queue or not.

The extTransition loops over all the in_ports, and then for every vessel in the inputs it's going to take it's destination such that we can see via our map_ports to which out_port we have to send it and we add it to the queue.

For the confluence port, we made a separate class because this is where a lot of analytics is done which was not necessary in the other confluences. This class is marked as the "start" and "end" of the port when vessels enter or leave. Checks are done whether the destination of the vessel is "S" which stands for sea. It also has 3 more out ports for statistics info, but the principle of the class remains the same.

Lock

atmoic_devs/lock.py
The lock has quite a lot of parameters. It needs to get all the information for the lock itself. With that information it can start calculating how long the gates stay closed and how long they stay open.

In the beginning they stay closed for 2 iterations without opening to not start at the hour.

The timer in this class works in three parts:

- Every hour it will make an update, so we have a timer that keeps the hour change
- It has a timer that keep track how long it needs to stay open or closed depending on the state
- It has a 30 second timer that is used when vessels are leaving as they need to leave 30 seconds apart.

The true challenge of this class is to make sure all the timers work with each other. The time advance function choses the shortest remaining time. And all the transition functions update all the timers.

The external transition will add incoming vessels to either the lock itself or a FIFO waiting list dedicated for either the sea or the dock input.

The internal transition will either:

- Update the hour for statistics.
- Let the next vessel move if there is one that can leave the dock. Then start the 30 second timer for the next to leave.
- Close the gate and start that timer.
- Open the gate on 1 side. Put the vessels that were in the lock into a list where they can leave 1 by 1. Put as many vessels in the lock as possible with FIFO. When vessels in front of the queue cant fit, but the vessels later in the queue can, they will enter the lock as well.

When testing we stumbled upon an extra problem: when we have too many tugboats in the lock. We can put too many tugboats in the lock at the same time so that the 30 second timer is too long and all the vessels cannot leave before the gates need to close. So we made the choice that we have a maximum number of vessels that can enter the lock at the same time. This maximum amount is defined by the time the lock stays open and the time between 2 leaving vessels.

Dock

atmoic_devs/dock.py
The state of the dock holds the quay_id, a dictionary of vessels with their remaining time, a list of vessels that are leaving and a list of requests to send.

The AtomicDEVS class has one in_port and out_port for vessels, as well as one in and out_port for messages/events.

The extTransition starts by updating the remaining times of the vessels. Then we take a look at all the input vessels and generate their wait time in the dock with a normal distribution. The vessels are to the dictionary with their waiting time as value.

The intTransition also starts by updating the remaining times. Then it checks if a vessel is ready to leave (remaining time is zero). Ifso it generates a request and adds it to the request list. This request is sent to the control tower via de out_event port

The timeAdvance function once again finds the shortest time between the vessels and gives no delay when a message or vessel needs to go.

Control Tower

atmoic_devs/control_tower.py
The control tower is in charge of communication for when vessels are trying to access or leave the port. It communicates with Anchorpoint to identify which vessels are allowed to enter and to which quay they must sail.

To do so we keep a list of docks, as well as their capacities. In the initializer of the ControlTowerState we loop over all the dock, and append 50. So we have a list that looks like [50, 50, 50, 50, 50, 50, 50, 50] where docks[0] gives the remaining capacity of dock 0.

There is also a queue which we use when all docks are unavailable. So when the confluence port sends a message to the control tower requesting a dock, this request is put in this queue. We will pop a request whenever a dock becomes available again.
In the AtomicDEVS class we have 2 in ports, one for incoming events and one, called free_event, which is used when a vessel leaves the dock. It also has one out_event to send to the anchorpoint to which quay a vessel must sail.

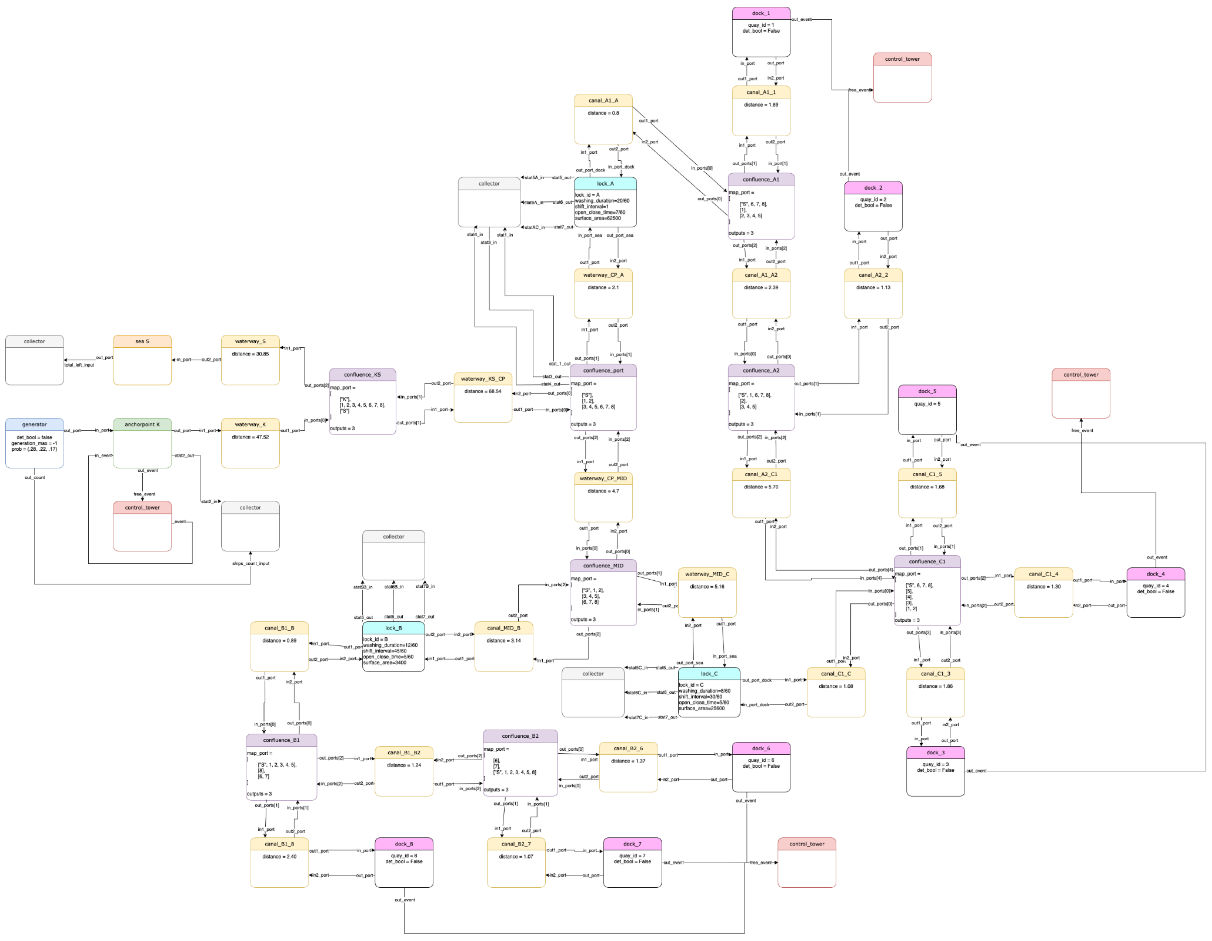
The extTransition checks if there are requests of vessels to enter the port. They are assigned to a dock if possible and the capacity of that dock is subtracted with 1. An answer message is also sent with a portEntryPermission. If there are no docks available, the request is added to the queue. It also checks if there are vessels who sent a message that they left the dock via the free_event port. In this case the capacity of the dock is increased with 1.

The intTransition checks if there is a place free and if there are requests waiting in the queue. If this is the case, we execute the same procedure as we did in the extTransition, by assigning it to a dock if possible. If no places are free, we don’t do anything here and add it back to the front of the queue.

The timeAdvance function returns 0 if there is an answer ready or if there is a place free in the queue. Else, this function reminds idle to wait.

We did not implement a smart load balancing port. The control tower will check from dock 1 to 8 if the dock is available and if so it will use that dock.

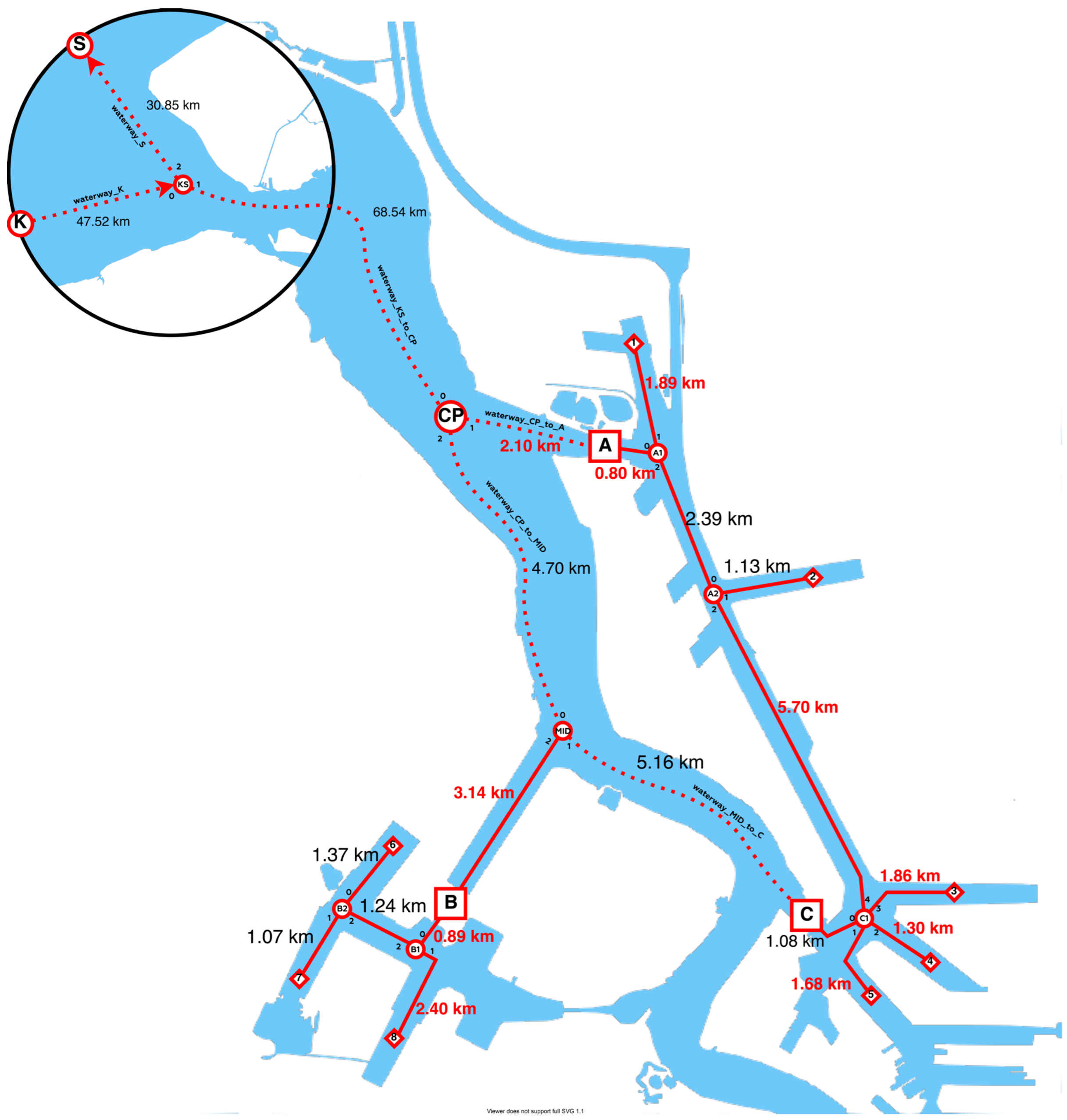
Task 2: Drawing of Model



In the image above you can see our complete system. This is most likely not easy to read at all, but such a zoomed-out view does give a good idea of the system as a whole and what is connected to what. In the resource map under the name **drawing_model.png** you can view the image in higher resolution so that you can easily zoom in on certain parts and see it in detail. There's not much explanation for this map needed because these are just all of our AtomicDEVS which are connected to the connectPorts function. What is visible however is that we have multiple instances of control_tower and collector. All instances of control_tower should really be viewed as 1 element. That you see multiple instances of this is purely to prevent us from drawing very long connections through our drawing. The same goes for the collector instance. In essence this drawing is just a copy of the map of the port but in python code.

Task 3: Coupled DEVS Model

In this task we simply implemented the map of task 2.
In the image below we've updated the map from the assignment such that it's visible which waterways we gave which names, and what confluences have which in and out ports. This is important for connecting the model properly. If we take a look at confluence CP, we see this is connected to 3 different waterways. The waterway_ks_to_CP is connected to the first input/output of the CP. When a vessel travels from the confluence KS to CP, it does so by using input port 0 on the Confluence CP. When a vessel wants to go from confluence CP to confluence KS, it uses the output port 0. The same principle holds for the other input and output ports for every confluence. This file can be found in the resources map under **port_confluence_numbers.png**.



In the model folder, we have a **model.py** file. Here we have a class called PortSystem which is a CoupledDEVS where we define all the atomic submodels we've created, and also connect all the ports needed to make the model work. This class also has generation_max parameter which can set the maximum number of vessels which are generated.

Task 4: Experimentation

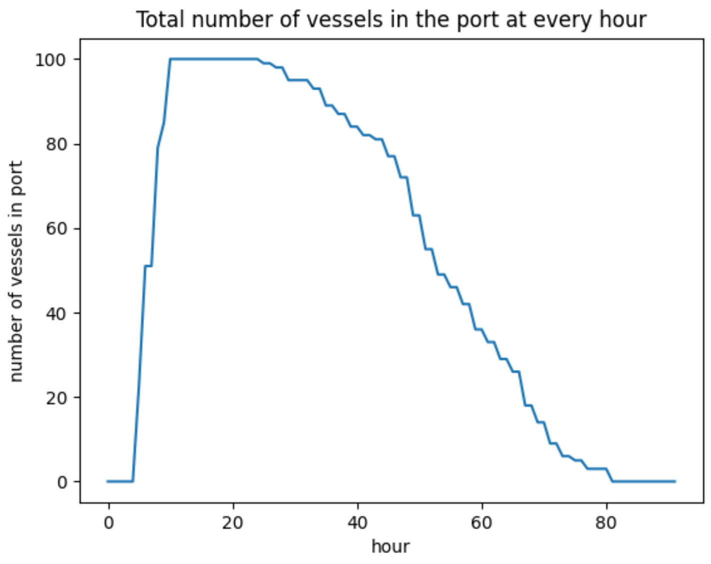
The **experiment.py** file sets the random seeds to the correct values, sets up the system, runs it with a certain termination time and gathers all the information from the collector to study the performance of the traffic in the port.

Task 5: 100 vessel generation

model/task5.py
All plots in this section can be found in: *model → plots → task5*. The txt file with all the information discussed below, but more detailed, can also be found in this folder.

When we run our program for generating 100 vessels, we see that 28% of these vessels are Crude oil Tankers, 23% are Bulk Carriers, 31% are Tugboats and 18% are Small Cargo Freighters. These probabilities are almost completely accurate with the probabilities we were given in the assignment. We ensured variation by sampling from an exponential distribution where we kept a list of scales of how many vessels are created each hour. Thus, the waiting time until we create a new vessel is sampled from Exp(λ), where 1/λ is the current average for that hour.

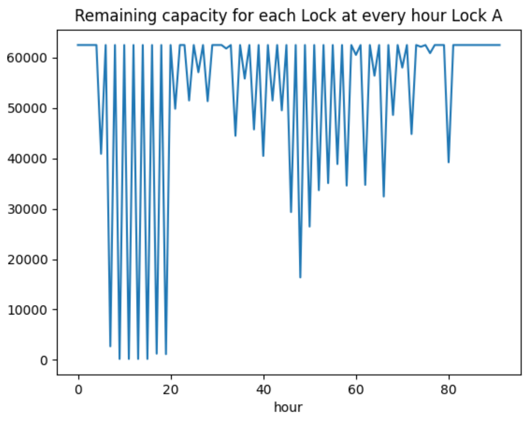
On average, there are about 45 vessels in the port. This is calculated with the following idea: when the number of vessels in the port changes it will add this new amount of vessels to a list. It will also add the time the previous amount of vessels was in the port to a list. This latter list will be used to take a weighted average of the first list.
The average travel time for a vessel is about 47.5 hours. With an average waiting time of 36 it means that a vessel is travelling for 11 hours in the port on average. A lot of this time is spent waiting in line before the lock. This is logical as all the 100 vessels are created in the first hour (on average) and travel through the same lock that has a Lock Shift Interval of an hour. Because there is a capacity of 400 in the docks and only 100 vessels, there is no wait time for a vessel to enter the port.In the graph below you can see another visual representation of the total number of vessels in the port at every hour.

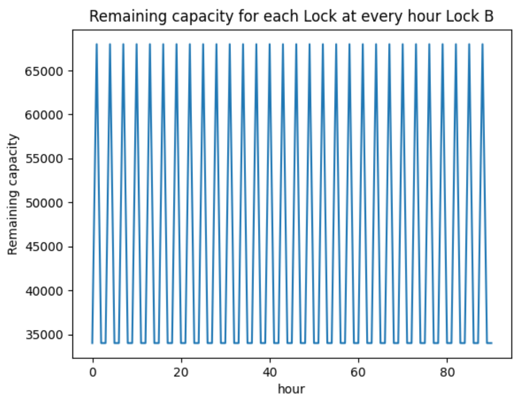


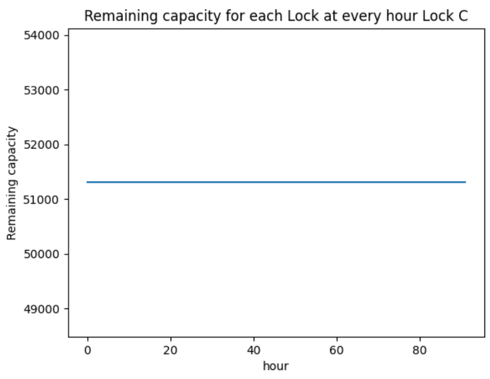
Here we see no results that we are surprised by, it rises fairly rapidly to 100, and from about the average vessels in the port it goes down fairly rapidly.

In this simulation there are 100 vessels generated, so lock B and C are not used. This is because we only use dock 1 and dock 2 because they provide enough capacity for 100 vessels and for these we pass through lock A. Therefore, the average idle time (time that there are no vessels inside) for lock B and C is equal to 1 hour, while the average idle time for A is about 14.7 minutes.

In the graphs below, we see the remaining capacities for Lock A, B and C every hour.







For B, the normal capacity of the lock is 34000. We see here that it alternates between 34000 and 68000, which is double. This can be explained because lock B has a lock shift interval of 45 minutes. So in the first hour we will have 1 lock shift with 1 full waisted capacity. In hour 2 we have 2 lock shifts (on minute 15 and on the hour change), here there are 2 fully waisted capacities.

With lock C, we just see a straight line at 51300 while the real capacity of this lock is 25650. The lock shift interval here is 30 minutes so you have 2 lock shift intervals per hour so the remaining capacity is double.

For lock A we see that in the beginning when the port has 100 vessels, that there is little to none remaining capacity available. As the simulation continues and no more vessels are generated and the first vessels start leaving the port, we see that the capacity grows bigger and bigger until it stays at maximum capacity at the end. At the beginning we see that the remaining capacity is only low on 1 side of the lock shift which is logical as no vessels are leaving yet.

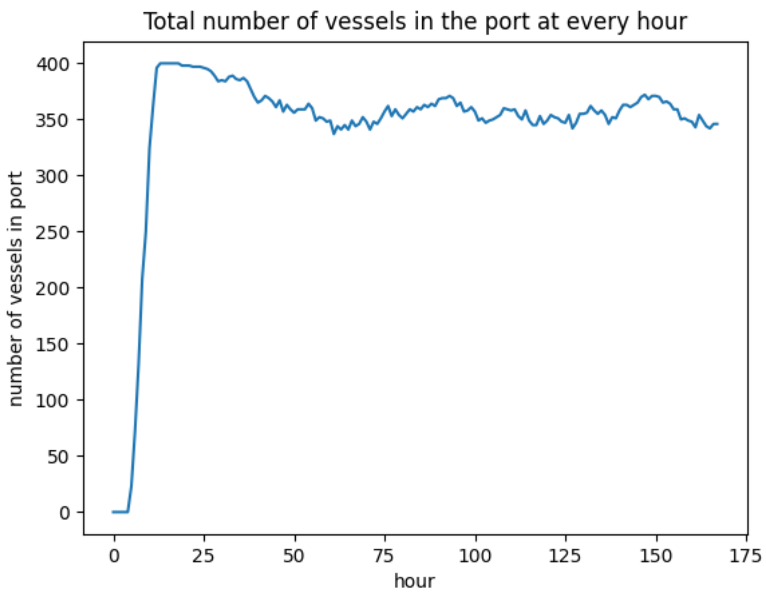
Task 6: 1 week simulation

Seed 42

model/tasks_seed42.py
All plots in this section can be found in:
model → plots → task6 → seed42.

When we run our program for 1 week (168 hours) with a seed of 42, we see that we've generated in total 25528 vessels. With probabilities that are very close.

On average, there are about 352 vessels in the port, the average travel time for a vessel is about 41.5 hours and a vessel has to wait in the anchor point for an average of 68.5 hours. Now with 25528 vessels generated it is very busy in the port. Thus waiting times start to run very high to enter the port. In the graph below you can see a visual representation of the total number of vessels in the port at every hour. The waiting is not surprising. Logically a que builds up when the capacity reaches its maximum. If we take the time between the first vessel that has to wait and the first vessel that leaves the port and add it with the average travel time we will come at a number around 68.

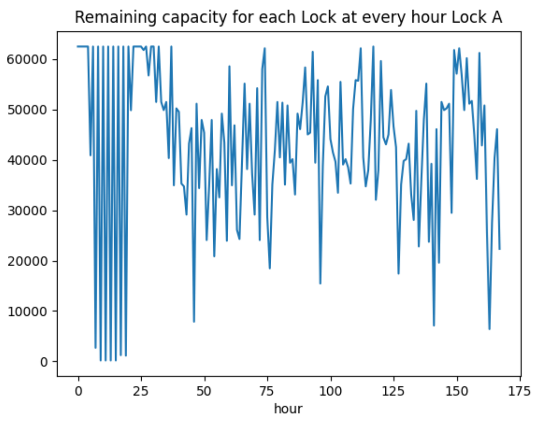


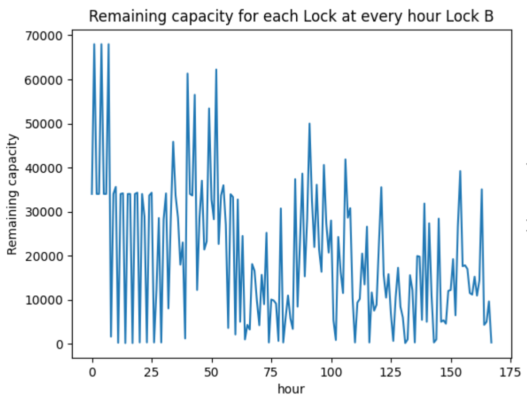
Here we see that it peaks at exactly 400, which is normal since we have 8 docks and each dock holds 50 vessels.

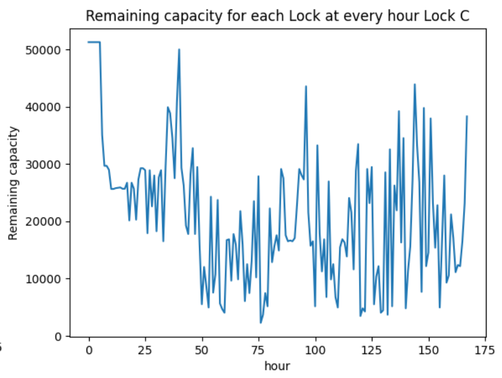
In the simulation where there were 100 vessels generated, lock B and C were not used because dock 1 and 2 had enough capacity. Now all 3 locks are being used because the port is running at full capacity. The average idle time for each lock is:

- Lock A: 0.07 hours
- Lock B: 0.04 hours
- Lock C: 0.04 hours

In the graphs below, we see the remaining capacities for Lock A, B and C every hour.







Seed 69

model/tasks_seed69.py
All the plots can be found in following folder:
model → plots → task6 → seed69.

When we run our program for 1 week (168 hours) vessels with a seed of 69, we see that there are absolutely no big differences at all in numbers. There is now an average travel time of 41.8 and an average waiting time of 68.7. But nothing major. This is expected as the random numbers are only used for the dock waiting time, initial delay of vessel creation and in choosing the vessel. But we work with pretty high numbers that all this will settle a bit to the average. The little improvements are for a large part there due to the increase in faster and smaller vessels.

Task 7: Alter constant values

All the plots can be found in following folder:
model → plots → task7

Change lock interval

In our PortSystem class we have a parameter change_lock_interval which has an initial value of 1. In the python file **model/task7_seed42_lock.py** you can see that we set the value here to ⅓. When we define our lock atomic submodels, we multiply this parameter with the given lock shift interval time given in the assignment. This has the effect that it lowers the amount of time it takes for a lock to complete 1 cycle. The time is taken away from the open time in a lock. So now lock A has an interval of 20 minutes, lock B has 15 minutes and lock C takes 10 minutes to complete.

Because entering a lock goes instantly and we saw that the lock did not use full capacity when it was running for a while in the precious task, we thought this would speed up the process. This was the case, we lowered the average travel time by about 1.5 hours in both seeds. While also reducing the average waiting time. Before you can think about implementing this, you should have a more detailed model that has a more accurate lock entering and leaving timing.

Change probabilities

model/task7_seed42_prob.py
Another way we altered the constants of our initial system is by changing the probabilities the vessels get generated. Although a tug boat may not be the fastest vessel, it is by far the smallest vessel of them all. This means that sometimes up to 30 more vessels can fit in a lock. This is important as waiting times in the lock is the biggest time waiter. We also then tried to increase the dock size, because there could fit more vessels in a lock, but this produced bad results. Although there may have been more vessels in the port on average, the average travel time for a vessel increased. This is due to the fact that there is way more traffic in the ports, and thus we opted not to change this parameter.

In our PortSystem class we have redefined all the probability values where there is a 91% chance of creating a tugboat, and a 9% chance of a Bulk Carrier. The Tugboats because they are small and the bulk carrier because they are fast. The other vessels are too big or slow to have a positive impact. This had an impact that both the average waiting time and the average travel time declined with more than 2 hours for both seeds.
This is ofc a drastic change in probability, but it shows that the type of vessels that enter the port have a drastic impact. Especially when you subtract the average waiting time in a dock from the travel time, its is an improvement of 6 to 4 hours on average.

Task 8: Deterministic Model

model/task8.py
All plots in this section can be found in:
model → plots → task8

To make our system go from a random model to a deterministic model, we had to change 3 aspects from our system.

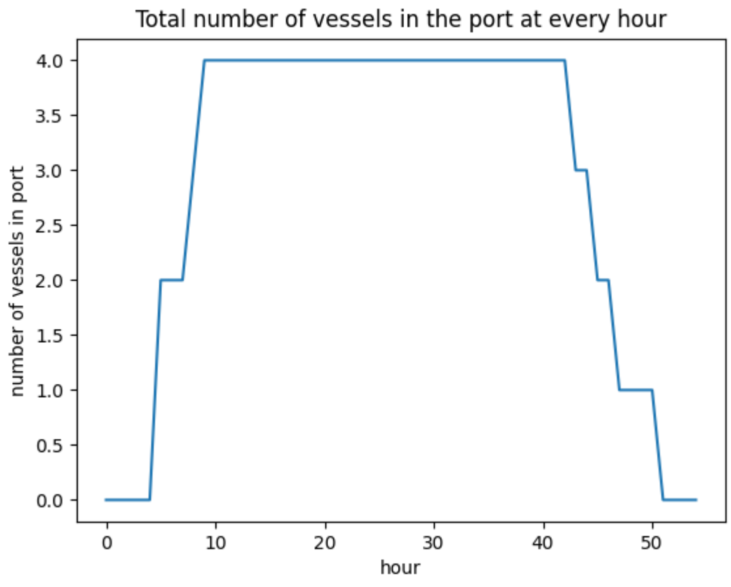
First the way the type of vessel is chosen. Right now, each vessel has a probability and we generate according to those probabilities. What we did is make a det_true variable that alters the way vessels are generated. Each generation of a vessel happens in some sort of cycle. First a tanker is generated, then a bulk carrier, then a tug boat and then a small cargo freighter. But we wanted to make it a bit more realistic than just generate an equal amount of each vessel type. So we keep a counter for each vessel type. These counters are reset to the probabilities on 100 vessels. Then we go in the cycle and every time we generate a vessel we lower the counter. When a counter reaches 0 it is skipped in all the following cycles. When 100 vessels are created, all counters are 0 and are reset again. This way we have no randomness but have some account for the probabilities.

Secondly we have the time between creations of vessels. This is just set to 1h divided by the average number of vessels that hour.

Lastly we have the waiting time in the dock. This is simply set to the average which is 36h.

We calculated the statistics, if only 4 vessels were generated for 55 hours:

We calculated that first that in the sixth hour both the Bulk Carrier and Crude Oil Tanker will arrive. Then in hour 9 the Tug Boat and in hour 10 the Small Cargo Freighter. This is also what we see in the following graph. This is calculated by dividing the distance to the port by the average speed of all the vessel types.

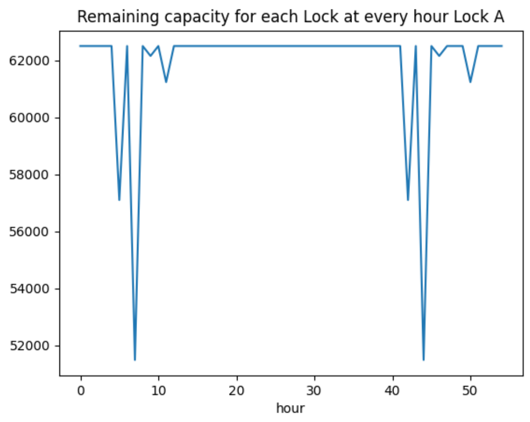


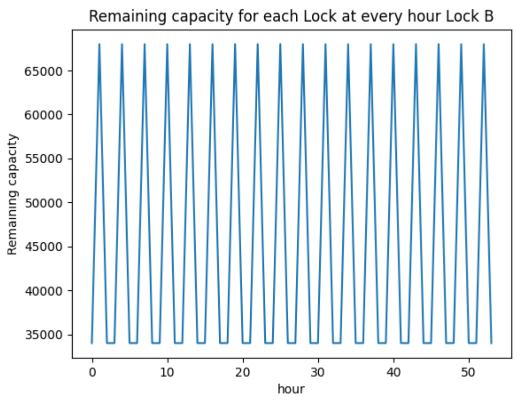
There should be no waiting time in the anchorpoint as there is more than enough room.

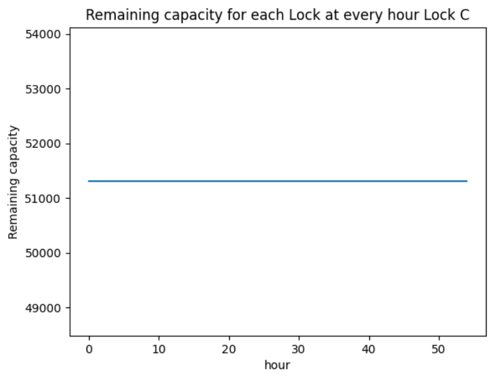
Only lock A should be used as all vessels will be assigned to dock 1.
To check this we can calculate the amount of cycles that the locks will take in 55 hours and see if they are all idle.

- Lock A has 55 cycles and 47 were idle. But There are 4 vessels that could use the lock, they all come in different cycles. So 8 cycles are not empty (2 way trip).
- Lock B has 73 cycles, all of them are idle.
- Lock C has 110 cycles, all of them are idle.

You can also see this on the following 3 images:







In Lock A you see the 6 cycles are not empty. But on the other locks it is always full remaining capacity. As a reminder in Lock B you sometimes have the case that 2 cycles take place in 1 hour and sometimes just 1.

Finally the total time for a vessel to make the complete trip, from entering until leaving the port is:
<u>**Bulk Carrier:**</u>
37.9955435558488
<u>**Crude Oil Tanker:**</u>
39.3825410595999
<u>**Tug Boat:**</u>
39.2244259837161
<u>**Small Cargo Freighter:**</u>
41.4887275011332

So the average would be: 39.5228095250745
Our implementation gave an average of: 39.53109675

So it's not perfect, we lost 0.01 hour somewhere in the implementation, this is only 36 seconds.
How we did this calculation can be found in the provided excel in the resource map under: **true_travel_time.ods.**