Reinforcement Learning with a large action space

Author:	Rens Hoogendorp
Date:	31 January 2023
TU/e supervisor:	Sem Borst
CQM supervisors:	Jan-Willem Bikker, Frans de Ruiter

This study was performed by Rens Hoogendorp during an internship at CQM as a master student mathematics from the TU/e (Eindhoven University of Technology).

RL CQM

Rens Hoogendorp

January 31, 2023

Tankwell
ANTI STATIC TANK CONTAINER
SUITABLE FOR PRODUCTS WITH A FLASH POINT < 60°C

Figure 1: Container used for transport

Contents

1 Introduction

In a logistic network where we move goods from one location to another location, there is almost certainly an unbalance. This means that the number of containers that are sent to a certain location might be greater than the number of containers that we want to send from this location. If we would not interfere, this would result in an accumulation of containers at that location. At another location we might have a structural shortage of containers. Then, it might make sense to sometimes send empty containers from the first location to the second location. However, it is not necessarily clear how many empty containers we need to send and when we would need to send them. This is because it costs money to send the empty containers, and it only generates money at a later stage. Furthermore, one cannot know for certain that the container is still needed, because demand is uncertain by nature.

In this report, we investigate how well Reinforcement Learning (RL) is suited for finding a good policy in this setting. (Deep) Reinforcement learning is a relatively recent development in Machine Learning. It has some particular terminology: an agent performs actions in an environment. Some of those actions are aimed at exploration (learning what would be best to do), some at exploitation (reaping the rewards of what has been learned). The goal is always to maximize total rewards, or rather, the cumulative reward over many steps. It does so by learning an optimal policy (i.e. prescription what action to take in each state). CQM has decades of experience of projects in supply chain logistics, where every day decisions are taken to optimize a cost / reward function, based on a model of the environment. There are good reasons to believe that CQM's projects are suitable for these techniques. However, getting it to work is often a craft involving research in techniques and creativity in designing the solutions.

We discuss the process of coming to a Reinforcement Learning agent in Section 3. We start with a simple RL algorithm. We discuss what the shortcomings of this algorithm are in our case and discuss improvements. We end up with a Reinforcement Learning algorithm that generates a policy in our logistic case. Then we quantify the policy our Reinforcement Learning agent produces in Section 4. We do this in different settings as we discuss in Section 4.0.1. We start simple setting where we can intuitively understand what an optimal policy would look like. Then we make the setting more complicated and create a setting based on data. We also test the robustness of the policy the Reinforcement Learning agent creates in Section 4.2. We do this by training on a certain setting and evaluate at another setting. Finally, we try to explain what the important aspects are for the policy generated by the Reinforcement agent in Section 4.3. This is interesting since RL is generally seen as a black box that generates a policy. However, it is not necessarily clear what aspects are important when creating this policy.

1.1 Contributions

The contributions of this report are;

- we show how to apply Reinforcement Learning techniques in an environment with large state space.
- we show how to reduce the complexity of the action space.
- we show how Reinforcement Learning can be used in existing organizational structures.
- we compare the resulting Reinforcement Learning policy with policies resulting from classical optimization techniques.
- we analyse the robustness of the Reinforcement Learning policy.

2 Model, notation, input and assumptions

We want to model a system where we have a given number of cities and a given number of containers. At each of these cities we get orders from an external source, these orders are a request to move a container to another city. We call these container movements "full container movements". When a full container arrives at a city, it is emptied and we can use this container again after some time. So, we move full containers from which we earn money. However, we cannot only move full containers (by fulfilling orders), because then we might be left with an unbalance in the system. The empty containers would accumulate at popular cities and as a result we can not fulfill orders at other cities. This is why we can also decide to move empty containers from cities to other cities, we call these movements "empty container movements".

We assume that the travel time between two cities is constant, so each link has its own travel time. We measure this travel time between two cities in full days. Each day we have one moment where we can send (full and empty) containers. We cannot fulfill an order if there is no container left at the city from which the order needs to depart. The orders arrive in the system according to some stochastic process. If we can fulfill an order we have to fulfill it. After we have fulfilled all orders we can decide whether we want to send some extra empty containers.

Finally, we have some sort of arrival process for external orders. The number of orders is follows form some distribution. We start with a simple Poisson distribution where the number of orders is independent over time and between links. Later we consider more complicated distributions like negative binomial, and correlations between links and over time.

2.1 Notation

We now define some notation that we use throughout the report.

$$C := \text{number of cities, where } C \in \mathbb{N}.$$
$$N := \text{total number of containers, where } N \in \mathbb{N}.$$
$$D := \text{travel time matrix between cities, where } D \in \mathbb{N}^{C\times C}.$$

Here $D_{i,j}$ denotes the travel time from city i to city j in full days. We make sure that all travel times adhere to the triangle inequality. Additionally, we assume that the travel time from city i to j is equal to the travel time from city j to city i.

2.2 Mathematical formulation

A RL agent can learn in a policy in a Markov Decision Process (MDP). A MDP consists of three main parts

- a state space,
- an action space,
- a reward system.

A policy function π is a mapping from the state space to the action space. When we fix a policy in an MDP we essentially have a Markov Chain. The goal in an MDP is to find a policy that maximizes the expected discounted reward.

Solutions for MDPs with finite state and action spaces can be found using dynamic programming based on value iteration[1]. Here, we first determine the optimal reward and then find the action that leads to this reward. Another method is policy iteration [2], in this method, we start with a policy and improve this policy step by step. We stop when the policy converges. Reinforcement Learning is a form of policy iteration.

In order to setup a RL agent which can start learning, we first need a to set up an MDP environment. As discussed, this environment consist of three main parts: a state space, an action space and a reward system. A state space is a set of elements, which we call states, where each element represents the information needed for an agent to make a decision. The action space consists of all possible actions that an agent can take. Finally, the reward system gives a reward based on the state transition and the action taken. Note that for a certain state action combination we do not always get the same reward. This is because there is randomness involved.

2.2.1 Action space

First, we look at which actions an agent can take. The agent has to decide how many (empty) containers we send between every pair of cities. So, we can represent an action as a $C \times C$ matrix. Therefore, the action space, $\mathcal{A}$, is a subset of $\mathbb{N}^{C\times C}$, i.e. $\mathcal{A} \subseteq \mathbb{N}^{C\times C}$. For example, let $A \in \mathcal{A}$. Then $A_{i,j}$ is the number of empty containers we send from city i to city j. Using this formulation we see that $|\mathcal{A}| = \binom{N+C^2-1}{C^2}$. This comes from the stars and bars problem [3]. We have at most N containers to place on C^2 links. However, note that not every action can be taken in each state, since there might not be enough empty containers in a city to send away.

2.2.2 State space

Second, we look at the state space. The state space is a set of states, where a state captures the important aspects of the system at every time step. Therefore, we start with identifying what information an agent needs to consider when deciding which action to take. All the information in the system is the location and status of each container. A first idea would be a size N-tuple, where each entry represents a container and contains its information.

Another formulation would be how many containers are at each location. The possible locations are not only the cities, but also the entire journey between cities A to city B combined with the information how long it has traveled. This formulation still has some redundant information, since the decision how many empty containers a city can "afford" to send away only depends on how many containers arrive at which time steps. It does not matter where the container came from. Note that the final decision of which action to take would most likely also depend on how much other cities "need" containers.

So in conclusion, the only information we need to capture is for each city, the number of containers that arrive in each coming time slot. This can be represented in a $C \times (1 + \max D)$ matrix. As a result, the state space $\mathcal{S} \subseteq N^{C\times(1+\max D)}$. For example, let $S \in \mathcal{S}$. Then $S_{c,d}$ denotes the number of containers that arrive at city c in d days. Note that $d = 0$ is a special case, since $S_{c,0}$ denotes the number of containers that are in city c. These containers are thus not moving and can be used to fulfill an order. Using this formulation we see that $|\mathcal{S}| = \binom{N+C\cdot(\max(D)+1)-1}{C\cdot(\max(D)+1)}$. This again is the stars and bars problem [3], we have N containers to distribute over $C \times (1 + \max D)$ locations.

2.2.3 Reward system

There are two kinds of rewards that we consider. The reward we get when we move an empty container, which is typically a negative reward. We also consider a reward when fulfilling an order, which is typically a positive reward.

Empty travel rewards are the rewards we get when we move an empty container. We assume that moving an empty container from city i to city j gives a reward of $-0.1 \cdot D_{i,j}$. We call the reward we get when fulfilling an order, the *full travel rewards*. We assume that when we fulfill an order that was from city i to city j, we get a reward of $0.5 \cdot D_{i,j}$.

The reward at time t is the positive rewards we gain from fulfilling orders minus the cost from the action we take.

3 Getting RL to work

In this section we describe the process of getting Reinforcement Learning to work. We discuss the main problems we encountered and how we solved them. Additionally, we discuss other potential solutions we tried and why we did not end up using them. We also mention some changes we investigated, some of these were useful and some of them not. The order in which we discuss this is roughly chronologically, i.e. in the order we encountered the problem. We start with a standard algorithm one would find in any Reinforcement Learning book [4], the pseudo code of a SARSA algorithm can be found in Algorithm 1. SARSA stands for State Action Reward State Action, this describes the iteration process we use in this algorithm. In this algorithm we make use of Q-values, a Q-value is an approximation of the expected discounted reward of a state and action pair. We build this SARSA algorithm up to the final version of the Reinforcement Learning algorithm we used.

The parameters of the Reinforcement SARSA algorithm are α, γ and ε. α is the step size, and we do not want this parameter to be too large because then it never converges to an optimal policy. On the other hand, we also do not want to take α too small because then the convergence would take a long time. γ is the discount factor, which implies how important the future is. We need this discount factor, because we have infinite horizon problem. This makes it necessary to discount the rewards. In our problem the future is quite important, therefore we fix γ at 0.9. Finally, ε is the probability of choosing a random action, rather than the default action. The default action is the action we think is the best, i.e. the action with the hightest Q-value. We do not want this to be too large, since we want to evaluate the actions which we think are best. ε should also not be too low, because sometimes we want to try actions which we currently think are not very good. This is desirable because we still want to explore those actions, as part of the exploration exploitation process [5]. We fix ε at 0.1.

Algorithm 1 SARSA algorithm

```
 1: Initialize parameters $\alpha, \varepsilon$ and $\gamma$
 2: Initialize Q-table
 3: for episode = $1, 2, \ldots$ Number of episodes do
 4:     Initialize environment
 5:     Choose action A from S using $\varepsilon$-greedy policy derived from Q-table
 6:     for run = $1, 2, \ldots$ Number of runs do
 7:         Take action A and observe reward R and state S'
 8:         Choose action A' from S' using $\varepsilon$-greedy policy derived from Q-table
 9:         Q(S,A) = Q(S, A) +$\alpha$(R + $\gamma$ Q(S', A')- Q(S, A))
10:         Set  S = S' , A = A'
11:     end for
12: end for
```

3.1 Large discrete state-action space

The first obstacle is the number of different states and actions we have in the current formulation. We saw that we have $\binom{N+C\cdot(\max(D)+1)-1}{C\cdot(\max(D)+1)-1}$ different states and we have $\binom{N+C\cdot(C-1))}{C\cdot(C-1)}$ different actions. As can be seen in Algorithm 1, for each combination of state and action, we want to approximate a Q-value. This means that we need to approximate $\binom{N+C\cdot(\max(D)+1)-1}{C\cdot(\max(D)+1)-1} \cdot \binom{N+C\cdot(C-1))}{C\cdot(C-1)}$ Q-values. This cannot be done in reasonable time for $C > 5$ and $N > 20$, since we have $\sim 10^{27}$ state-action pairs.

Note that we consider each state completely on its own. This means that the Q-values for two states that might "look" similar are computed separately from each other. For example consider the following two states

$$S_1 = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 0 & 0 \\ 2 & 0 & 1 \end{pmatrix} \text{ and } S_2 = \begin{pmatrix} 2 & 1 & 0 \\ 3 & 0 & 0 \\ 2 & 0 & 1 \end{pmatrix}. \tag{1}$$

In this example we have three cities and nine containers. In state S_1 we have 1 container at city 1, 2 containers 1 day of travel removed from city 1, 3 containers at city 2, 2 containers at city 3 and 1 container 2 days removed from city 3. When we compare this to state S_2 we see that the situation is exactly the same for cities 2 and 3. The only difference is that there are 2 containers at city 1 instead of 1, and that there is 1 container 1 day removed from city 1 instead of 2. Intuitively it make sense that, if an action has a high Q-value for S_1 it would also have a high Q-value for S_2 and vice versa. The same is true for actions with a low Q-value.

To use this observation, we use an Episodic Semi-gradient SARSA algorithm. The Episodic Semi-gradient SARSA algorithm is a parametric model instead of a model based on tabulation. This means that instead of computing a Q-value for each state action pair, we now create a Q-function. This Q-function has as input a state and an action, and returns a Q-value. This Q-function has a fixed number of parameters and the output depends on these parameters, the state and the action. In an Episodic Semi-gradient algorithm we update the parameters as can be seen in Algorithm 2.

Algorithm 2 Episodic Semi-gradient SARSA algorithm

```
1: Initialize parameters $\alpha, \varepsilon$ and $\gamma$
2: Initialize parameters, $w$, of Q-function
3: for episode $= 1, 2, \dots$ Number of episodes do
4:     Initialize environment
5:     Choose action A from S using $\varepsilon$-greedy policy derived from Q-function
6:     for run $= 1, 2, \dots$ Number of runs do
7:         Take action A and observe reward R and state S'
8:         With probability $1 - \varepsilon$ let A' = $\text{argmax}_{a \in \mathcal{A}} \hat{Q}(S', a, w)$, i.e. the action derived from the Q-function
9:         With probability $\varepsilon$ let A' be another action acording to some policy.
10:        $w = w + \alpha(R + \gamma\hat{Q}(S', A', w) - \hat{Q}(S, A, w))\nabla\hat{Q}(S, A, w)$
11:        Set  S = S' , A = A'
12:    end for
13: end for
```

The choice for the Q-function we use is not trivial. Also, the function prediction space, (S, A), is highly dimensional and is therefore not easy to visualize. Later we return to the issue of choosing the Q-function.

3.2 High dimensionality

The second big obstacle occurs when we apply the Episodic Semi-gradient SARSA algorithm with some Q-function. We see in Algorithm 2 line 7 that we need to find $\text{argmax}_{A \in \mathcal{A}} \hat{Q}(S, A, w)$ in each iteration with some probability. Since we want to train the agent for a long time, i.e. with a high number of iterations, it is key that we can solve this optimization problem quickly.

Note that the action space $\mathcal{A} \subseteq \mathbb{N}^{C \times C}$. This means that we need to solve a C^2 dimensional optimization problem. As it is simply a difficult problem, since there might be a lot of local maxima/minima and saddle points as discussed in [6]. Therefore we want a more efficient way to find the best action possible, given the state. This is a problem not usually discussed in literature, most action spaces have a low dimensionality [7], [4], [8]. There are a few ways to do this, however we will consider local agents instead of a global agent.

3.2.1 Local agents

We now explain what we mean by local agents. Up until now, we used one agent with as action space a matrix which represents how many containers we send on which links. The idea behind local agents is that each local agent represents one city. The action a local agent takes is defined by how many containers the corresponding city wants to send/receive. We can make this one-dimensional and we interpret wanting to send $-x$ containers as wanting to receive x containers. The dimension of the action space is then no longer $C \times C$, but instead we have to consider C one-dimensional action spaces.

Using this logic we now consider C local agents, where each agent represents one city. Note that each local agent has its own Q-function with its own parameters. On the other hand, the local agents share the state space with the full information available to each agent. We denote the local Q-function of city c by $\hat{Q}_c(S, a, w_c)$ with $S \in \mathcal{S}$. The local action space is now simply one number $a \in \tilde{\mathcal{A}} \subseteq \mathbb{Z}$, where $a > 0$ means city c sends a containers while $a < 0$ means city c receives $-a$ containers. Finding the optimum of this function is much easier since it is only an one-dimensional optimization problem.

Since we now train C agents simultaneously, we also need to distribute the rewards and costs in some way, since it is not evidently clear how the local agents would interact with the reward system. One choice could be to simply give the total reward in one day to each agent. This does seem fair since there is a large influence from different cities on the reward. This is why we divide the reward in such a way that city c only gets rewards and costs where city c is directly involved in. When we move an empty container from city c to city A, we divide the cost over both those corresponding agents. Similarly, when we can fulfill an order from city c to city B, the reward is again divided over both these corresponding agents.

Another argument for using local agents is that this structure more closely represents the structure in the company, where these decisions are typically made. Each city/terminal has a local terminal manager who is in charge of that terminal. To determine how the empty containers are sent, all the terminal managers come together to discuss this. Each terminal manager only knows if that terminal has a preference to receive containers or can send containers away.

3.2.2 Mapping local actions to global actions

Now we have an easy way to optimize (and evaluate) local Q-functions. However, we still need a (global) action for the environment. This is why we need a way to map these local actions to a global action. This mapping represents the meeting where the terminal managers come together to decide which terminal has to send containers to which terminal.

The first idea is to do this with a max flow min cost algorithm. A max flow min cost algorithm finds maximum size flow from a source to a sink. When there are multiple flows of maximum size it selects the one with lowest cost. The input of a max flow min cost algorithm is a graph with nodes and edges. Each edge has a capacity and a cost, the cost is per unit sent. The flow that the algorithms outputs does not exceed the capacity at any edge, is preserved in every node except the sink and source (i.e. the flow into a node is the same as the flow from a node), the flow has maximum, size and the flow has minimum cost. The cost that we want to minimize is the total travel time.

To apply this algorithm to our mapping problem, we create a graph with $C+2$ nodes as in Figure 2. The first C nodes represent the different cities in the problem and the last two nodes are the source and sink respectively. From the source, we add an edge to every vertex that corresponds to a city that wants to send containers, with capacity equal to the number of containers that city wants to send and cost 0. From each vertex that corresponds to a city that wants to send containers, we add an edge to vertices of cities that want to receive containers, the cost of this edge is the travel time between the corresponding cities. On these edges there is no capacity, since the restricting factor of how many containers can be sent from a city equals the number of containers are available in that city. So, for example the edge between sending city i and receiving city k has an unlimited capacity and a cost of $D_{i,k}$ (since the cost/reward is linearly proportional to the distance between two cities). Finally, we add from each vertex that corresponds to a city that wants to receive containers an edge to the sink with cost 0 and as capacity the number of containers that city wants to receive.

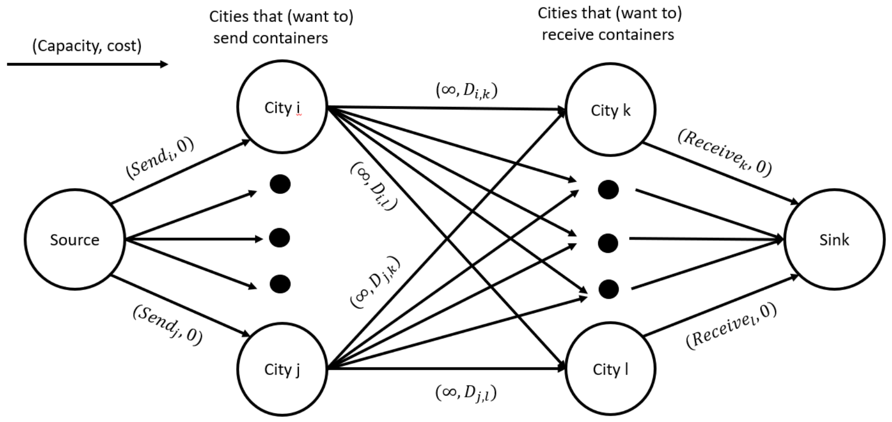

Figure 2: A visualisation of the instance we use for the min cost max flow algorithm. $Send_i$ is the number of containers that city i wants to send and $Receive_l$ is the number of containers that city l wants to receive.

The problem with this idea is that we first fix the number of containers each city wants to send/receive, and based on this how the containers need to be sent. As a result, the total number of containers that the terminal managers want to send might not be equal to the number of containers that they want to receive. If this is the case, the links that are used are the links with the lowest cost. Only after all the shortest links are filled, we start using costlier links. This will result that cities which are far away will structurally receive fewer containers. On the other hand, cities which have short connections to most cities will almost always send/receive containers.

To tackle the problem of unbalanced sending, we also want to take into account how much a city prefers to send/receive containers. The reward we expect to gain when we receive/send a container is entirely captured in the Q-values. To make the flows more balanced, we give the Q-values also as input of the max flow min cost algorithm. We have two stages in the algorithm, the first one determines the number of containers each city is going to send/receive based on the Q-values. We replace the edges between cities in Figure 2 by multi-edges. Each edge of the multi edge represents one container, and has capacity 1. The cost of this edge is the increase/decrease in total sum of Q-values as result from sending that container. In this multi-graph we find a max cost flow. This flow represents the maximal increase in total Q-value with the constriction that the total number of containers sent is equal to the total number of containers received. Then we use the same strategy as before to determine which cities send their containers to what destination.

3.2.3 Local Q-functions

We now discuss the local Q-functions in more detail. First, we look at some of the aspects we expect such Q-functions to have. Then we propose a framework, which we work out and analyse. After that, we discuss some small issues and ideas that improve the performance.

When we focus on one city and look at its expected discounted reward we have some observations. Sending two containers away, is worse than sending one container away for a city. Sending three containers is even worse for that city. This works the other way around and is also true for receiving containers. When we translate this to Q-values and Q-functions we expect that the local Q-function is concave, with respect to the action.

Because of this we use a concave parabola, since this is one of the simplest concave functions. So, we fit a function of the form

$$f(x) = b_0x^2 + b_1x + b_2, \text{ where } b_0 < 0 \text{ and } x \text{ is the number of containers send/receive.} \tag{2}$$

The parameters b_0, b_1 and b_2 of the Q-function function depends on the state. An easy start is to let them depend linearly on the state. Putting this all together gives the following Q-function for city c

$$\begin{aligned} Q_c(S, a, w^c) = &\left(w^c_{0,0} + \sum_{l=1}^{C} \sum_{d=0}^{\max(D)} w^c_{0,(l,d)} \cdot S_{l,d} \right) \cdot a^2 \\ &+ \left(w^c_{1,0} + \sum_{l=1}^{C} \sum_{d=0}^{\max(D)} w^c_{1,(l,d)} \cdot S_{l,d} \right) \cdot a \\ &+ \left(w^c_{2,0} + \sum_{l=1}^{C} \sum_{d=0}^{\max(D)} w^c_{2,(l,d)} \cdot S_{l,d} \right). \end{aligned} \tag{3}$$

As stated before, we want $\left(w^c_{0,0} + \sum_{l=1}^{C} \sum_{d=0}^{\max(D)} w^c_{0,(l,d)} \cdot S_{l,d}\right)$ to be negative. For the Q-function of city c, $(w^c_{k,l,d})_{k=0,1,2;l=1,\dots,C;d=0,\dots \max(D)}$ are the parameters we want to estimate. Using this formulation we need to estimate $3 \cdot C \cdot (\max(D) + 1))$ parameters for each local agent.

There are multiple ways to parameterize a parabola and estimating the coefficients b_0, b_1, b_2 directly. While the parameterization is easy, it might not be the best idea. For example, the horizontal location of the maximum of the parabola has an intuitive interpretation in this context. Namely, this is the best number of containers to send/receive for that city. Because of this, it makes sense to estimate the horizontal location of the top of the parabola, i.e. $-\frac{b_1}{2b_0}$. Let $b_3 = -\frac{b_1}{2b_0}$. The coefficient b_0 represents how costly sending/receiving another extra container is. Finally, the coefficient b_2 represents how good not sending/receiving any containers is. Therefore, we estimate (b_0, b_3, b_2) instead of (b_0, b_1, b_2). Using this the local Q-function of city c is now

$$\begin{aligned} Q_c(S, a, w^c) = &a \cdot \left(w^c_{0,0} + \sum_{l=1}^{C} \sum_{d=0}^{\max(D)} w^c_{0,(l,d)} \cdot S_{l,d} \right) \cdot \left(a - 2 \cdot \left(w^c_{1,0} + \sum_{l=1}^{C} \sum_{d=0}^{\max(D)} w^c_{1,(l,d)} \cdot S_{l,d} \right) \right) \\ &+ \left(w^c_{2,0} + \sum_{l=1}^{C} \sum_{d=0}^{\max(D)} w^c_{2,(l,d)} \cdot S_{l,d} \right). \end{aligned} \tag{4}$$

Note that we still estimate the same number of parameters.

To reduce the number of parameters we want to estimate we make the following observation. When we look at the state at a certain point in time we can expect that most of the containers are close to or in cities. As a result, we expect that the values of the first few columns of the state matrix to be significantly higher than the values in the last columns. Using this we can summarize the state matrix and thus reduce the number of parameters. For $d = 4, 5, \ldots$ we use the same weight being $w_{k,(c,\min(4,d))}$. We model (b_0, b_3, b_2), by $(h_0(S), h_1(S), h_2(S))$, where

$$h_k(S) = w_{k,0} + \sum_{c=1}^{C} \sum_{d=0}^{\max(D)} w_{k,(c,\min(4,d))} \cdot S_{c,d}, \text{ for } k = 0, 1, 2 \text{ and } S \in \mathcal{S}. \tag{5}$$

Using this formulation we now have $3 \cdot C \cdot 5$ parameters per local agent.

3.3 Initialization and numerical stability

Lastly, we discuss some more generic problems we encountered. Most of these have to do with numerical stability and its influence at finding an optimum. For example, scaling of the parameters is very important for numerical stability. Additionally, initializing is important for finding a stable optimum. Finally, we used a n-step SARSA algorithm to improve our estimates for the discounted reward.

3.3.1 Initializing

When we look at a single Q-function, we have a large number of parameters that influence the Q-value per action in different ways. We cannot initialize all these parameters randomly and expect that the reinforcement agent can find a good/better policy. For example, we expect that the parameters that determine the concavity of the parabola, the b_0 coefficient, should not be negative. Also some specific combinations of parameters might result in the agent never converging, or even diverging.

To prevent this behaviour, we initialize the parameters of the Q-function. More specifically, we initialize for each city c $w^c_{0,0}, w^c_{0,1}$ and $w^c_{0,2}$ at specific values, and all other parameters at zero. With the initialization we want to help the agent in the right direction, but not give such a bias that the final policy is almost determined by the initialization. This is why we set all the parameters that are multiplied with an entry from the state matrix to zero. This way the policy yields the same action for each state at the beginning.

To determine the values for $w^c_{0,0}, w^c_{0,1}$ and $w^c_{0,2}$, we run two simulations. With the information we get from the first simulation we determine $w^c_{0,2}$. Then with the information we get from the second simulation we determine $w^c_{0,0}$ and $w^c_{0,1}$. In the first simulation we run the environment, however we take no actions. This gives an estimate for the expected discounted reward for each city when every city does not send/receive any empty containers. This will then be the value of $w^c_{0,2}$.

The second simulation is more complicated. We first determine the number of empty containers every city needs to send each day, such that the expected inflow is equal in expectation to the expected outflow. This number for city c is the value of $w^c_{0,1}$. Then we run a simulation where city c sends/receives $w^c_{0,1}$ containers every time. This gives an estimate for the expected discounted reward for each city when every city sends/receives $w^c_{0,1}$ empty containers. We call this reward of city c M_c, then we initialize the value $w^c_{0,0}$ as follows: $w^c_{0,1} = -\frac{M_c - w^c_{0,2}}{(w^c_{0,1})^2}$.

The idea behind this procedure is that we fit a parabola through some points and then translate this parabola to a local Q-function in such a way that the policy that follows is state-independent. We fit the parabola using the following two points, the place of the top and its value and the value at 0. Together this gives enough information to fit an unique parabola.

3.3.2 Scaling

As we saw before, we can categorize the parameters of one local Q-function in three types. The first type has effect on the b_0 coefficient, the second type is the group of parameters that determine the location of the top, and the last type affects the b_2 coefficient of the parabola. The b_0 and b_2 coefficient and the location of the top all have different interpretations. As it turns out, all these have different order of magnitude. As a result, all the parameters of the local Q-function have a different order of magnitude. As a result, we might want to use a different step size α for each different type of parameters.

To solve this problem we use per local agent three normalization constants, one for each type. Using the initialization we did, we scale the parameters in such a way that the mean of the scaled parameters are all of the same order of magnitude. We call these normalization constants Q_0, Q_1 and Q_2. Then the local Q-function of city c looks like

$$\begin{aligned} Q_c(S,a,w^c) =& Q_0 \cdot \left(w^c_{0,0} + \sum_{l=1}^{C} \sum_{d=0}^{\max(D)} w^c_{0,(c,\min(4,d))} \cdot S_{l,d} \right) \cdot a^2 \\ & - 2 \cdot Q_0 \cdot \left(w^c_{0,0} + \sum_{l=1}^{C} \sum_{d=0}^{\max(D)} w^c_{0,(c,\min(4,d))} \cdot S_{l,d} \right) \cdot Q_1 \cdot \left(w^c_{1,0} + \sum_{l=1}^{C} \sum_{d=0}^{\max(D)} w^c_{1,(c,\min(4,d))} \cdot S_{l,d} \right) \cdot a \\ & + Q_2 \cdot \left(w^c_{2,0} + \sum_{l=1}^{C} \sum_{d=0}^{\max(D)} w^c_{2,(c,\min(4,d))} \cdot S_{l,d} \right). \end{aligned} \tag{6}$$

3.3.3 SARSA with n-step

Finally, we use n-step SARSA algorithm. As we saw before in Algorithm 2 in line 9. We update the parameters w using the difference between the expected discounted reward in state S and the reward in the next step plus γ times the expected discounted reward of the next state. The second part is a better estimate for the expected discounted reward in state S. We can improve this estimate even more by looking n step into the future instead of only one step. For this we use an algorithm that is called the n-step SARSA algorithm.

When we combine all improvements steps we get Algorithm 3 for finding a policy.

Algorithm 3 Episodic Semi-gradient n-step SARSA algorithm with local agents

```
1: Initialize parameters $\alpha, \varepsilon, \gamma$ and $n$
2: Initialize the values of $(w^c_{0,0}, w^c_{0,1}, w^c_{0,2})$ for each local agent
3: Scale the parameters by initializing $(Q_0, Q_1, Q_2)$ and rescale $(w^c_{0,0}, w^c_{0,1}, w^c_{0,2})$ for each local agent accordingly
4: for episode $= 1, 2, \ldots$ Number of episodes do
5:     Initialize environment
6:     With probability $1 - \varepsilon$ let $A_0$ be the global action derived from the local Q-functions and the min cost max flow algorithm
7:     With probability $\varepsilon$ let $A_0$ be another global action according to some policy.
8:     Determine the local actions $(a_{c,0})_{c=1}^{C}$ from global action $A_0$
9:     for $t = 0, 1, 2, \ldots$ Number of runs do
10:        Take global action $A_t$ and observe local rewards $(R_{c,t+1})_{c=1}^{C}$ and state $S_{t+1}$
11:        With probability $1 - \varepsilon$ let $A_{t+1}$ be the global action derived from the local Q-functions and the min cost max flow algorithm
12:        With probability $\varepsilon$ let $A_{t+1}$ be another global action according to some policy
13:        Determine the local actions $(a_{c,t+1})_{c=1}^{C}$ from global action $A_{t+1}$
14:        $\tau = t - n + 1$
15:        if $\tau > 0$ then
16:            $G = \left(\sum_{i=\tau+1}^{\tau+n} \gamma^{i-\tau-1} R_{c,i} + \gamma^n Q_c(S_{\tau+n}, a_{c,\tau+n}, w^c)\right)_{c=1}^{C}$
17:            for $c = 1, \ldots C$ do
18:                $w^c = w^c + \alpha(G_c - Q_c(S_\tau, a_{c,\tau}, w^c))\nabla Q_c(S_\tau, a_{c,\tau}, w^c)$
19:            end for
20:        end if
21:    end for
22: end for
```

4 Results and evaluation

We now evaluate the performance of the agent. For this we use two instances. An instance is a combination of the number of containers, the number of locations and the distances between those locations. Furthermore, we use different arrival processes, which we refer to as environments. The first instance is quite a simple setting, here we can intuitively understand what an optimal policy would look like. The second instance is based on data, and is more complex. Since the instance is based on data it also closely resembles a real-life situation.

4.0.1 Instances

The first instance has $C = 10$ cities and $N = 450$ containers. All cities lie on a horizontal line, in such a way that the distance between two cities equals 2 plus the number of cities between those cities. The instance is visualized in Figure 3. The blue dots are the cities and the size of the dot is determined by the outgoing rates. The size of the arrows is in proportion with the rate of that link.

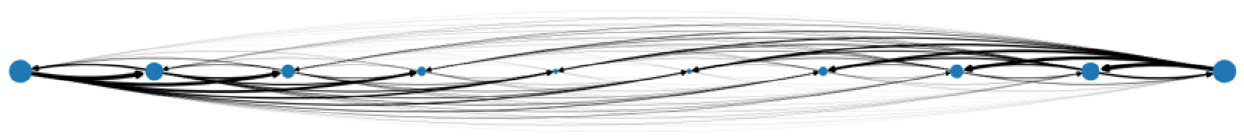

Figure 3: The first instance

We construct the instance such that the outer cities have a higher output. Therefore, we expect that on average more empty containers need to be sent to the outer cities. We discuss this later in more detail.

The second instance we create is based on the container usage in Europe. We have some data about The data corresponds to the movements of containers in a few days. The data set contains different information, the most important ones are between which two cities the movement was and how long it took to travel between the two locations. Since we do not have a large amount of data, we group cities by country. Therefore, we refer in this instance to locations instead of cities. After this grouping, there are still some links that have never been used in the data. We use a regression to fill in the missing travel times. In this regression we fit the travel time based on the physical distance. Since the travel times should adhere to the triangle inequalities, we adjust some problematic travel time between cities such that they adhere to the triangle inequality. This results in the instance in Figure 4. Here the size of the arrows represents the average number of orders between those two countries.

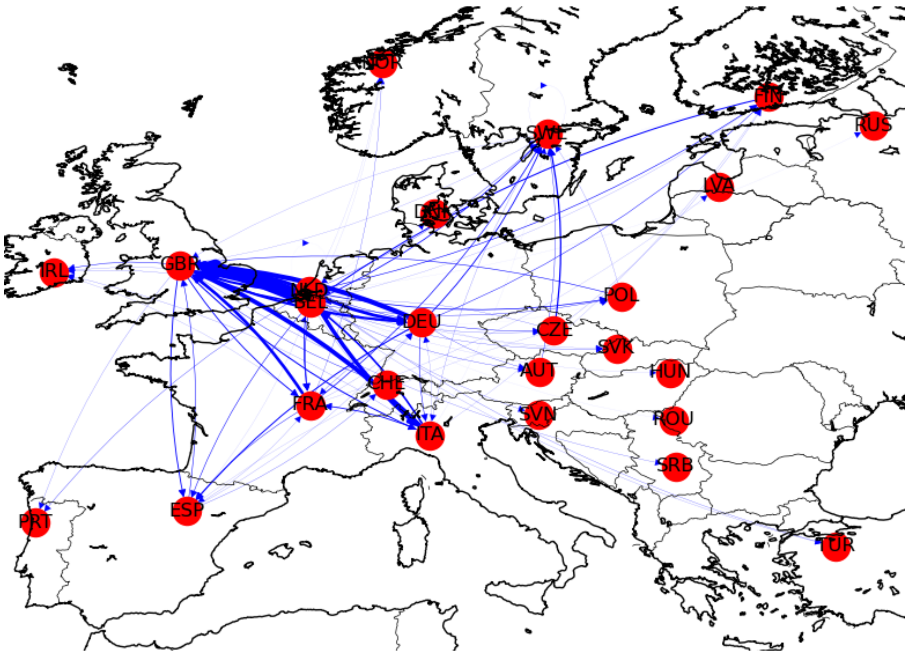

Figure 4: The instance based on data

This results in an instance with 25 countries and a distance matrix, which we can use to create our environment. Additionally, we know the average number of orders over time on each link. Based on this we construct our arrival process. We fix the number of containers in such a way that the average usage of a container over time is on average approximately 80%. This means that on average 20% of the containers is stationary at the locations and 80% is traveling (either empty or full). It turns out that we need around 15000 containers to achieve this.

4.0.2 Environments

As stated before, based on the travel count per link we fit an arrival process. We start with an arrival process for each link where all the number of arrivals on each day and link are independent Poisson distributed with rate matrix λ. $\lambda_{i,j}$ is the rate parameter for the Poisson distribution of link (i,j). $\lambda_{i,j}$ is the average travel count on that link. A key principle of the Poisson distribution is that the standard deviation equals the square root of the mean. As a result, the spread is relatively low. Therefore, we also consider a negative binomial, where we set the standard deviation equal to the mean squared. Finally, we consider an environment where all arrivals are again Poisson distributed. However, now we change the rates each episode. We do this by sampling a random variable for each link and multiply this with the original rate. This random variable has mean 1 and some variance. The specific distribution we use is a log normal.

Note that there is no correlation between links and over time in any of these environments. In a real-world setting one might expect that there is correlation between links and over time. We consider two different correlations. The first one is correlation over time, for this we use an AR-1 structure [9]. The second correlation we consider is between countries. For this we divide the countries in five groups, namely, Scandinavia, British Isles, Western Europe, Southern Europe and Eastern Europe. We denote with $g(i)$ the index of the group of country i. The last environment we consider incorporates these correlations in the following way:

$$\begin{aligned}
&\lambda_{(i,j),t} = c_{i,j} + B_{i,t} + E_{j,t} + F_{g(j),t}\\
&B_{i,0} \sim N(0,\sigma_0^2), E_{i,0} \sim N(0,\sigma_1^2), F_{i,0} \sim N(0,\sigma_2^2),\\
&B_{i,t} = \rho_0 \cdot B_{i,t-1} + \Sigma_{i,t}, \text{ where } \Sigma_{i,t} \sim N(0,\sigma_0^2),\\
&E_{j,t} = \rho_1 \cdot E_{j,t-1} + \Phi_{i,t}, \text{ where } \Phi_{i,t} \sim N(0,\sigma_1^2),\\
&F_{g(j),t} = \rho_2 \cdot F_{g(j),t-1} + \Theta_{i,t}, \text{ where } \Theta_{i,t} \sim N(0,\sigma_2^2),\\
&X_{i,j,t} \sim Poisson(\max(\lambda_{(i,j),t}, 0)).
\end{aligned} \tag{7}$$

Here, $X_{i,j,t}$ is the number of orders from city i to city j on day t. The parameters of this environment are $\{c_{i,j}\}_{i,j\in\{1,\ldots,C\}}$, $\{\rho_i\}_i = 0^2$ and $\{\sigma_i^2\}_i = 0^2$.

We tried to fit these parameters based on the data. we used several methods, including mixed effects Poisson regression using maximum likelihood and using Bayesian estimation. Unfortunately, the fits were not great. The variances, $\{\sigma_i^2\}_i = 0^2$, were all very large which has as a result that the total number of orders per day was much higher on average. This may be explained by the fact that we only have data of a few days. Another explanation is that the model is simply not the right model to explain the data. Therefore, we choose these parameters in such a way that makes sense within the setting and such that the average number of orders per day is the same. For $\{c_{i,j}\}_{i,j\in\{1,\ldots,C\}}$ we use the average travel count for each link in the data. We expect a positive correlation over time, thus we use $\rho_0 = \rho_1 = \rho_2 = 0.8$. This has as effect that we have periods where we have a larger number of orders. On the other hand, we also have periods where we have less orders. Finally, we want the effect from the correlation terms to be significant but not be the dominating factor. After some experimentation, this it turns out that $\sigma_0^2 = \sigma_1^2 = \sigma_2^2 = 1$ are good choices.

We want to compare these environments in some way. We look at the variance divided by the expected value of each link in an environment. We express this in the average link count, λ. For example, in the Poisson environment the variance divided by the expected value is simply one. In the Negative Binomial environment we choose the parameters in such a way that this ratio is λ. The variance of the moved Poisson can be computed using the fact that

$$Var[XY] = \mathbf{E}[Y]^2 Var[x] + Var[Y]\mathbf{E}[X^2], \tag{8}$$

when X and Y are independent random variables. The variance of the correlation model is complicated since the parameter of the random variable is also a random variable itself. Here we use the fact that

$$Var[Poi(a+X)] \simeq a + Var[X]. \tag{9}$$

This approximation becomes closer to an equation when the variance of X is smaller. When we look at Table 1, we see that Poisson, Moved Poisson small and the Correlation model all have a relative small variance expected value ratio. The ratio of the other two environments are much larger. This is especially true for λ around 100.

Environment	Variance/expected value
Poisson	1
Negative Binomial	λ
Moved Poisson small	1.01 +0.01$\cdot\lambda$
Moved Poisson big	1.1 + 0.1$\cdot\lambda$
Correlation model	$\sim$1 + $\frac{3}{\lambda}$

Table 1: The ratio of variance and expected value per link in terms of the average travel count, λ

4.0.3 Policies

We want to compare the policy that the Reinforcement Learning agent returns. We do this by considering other policies and evaluating them at the same instance. Some of these policies are simple in nature, while others are more complex. The simplest one is never sending any containers, we refer to this policy as Doing nothing.

Another policy is taking random actions. Here we do not send containers with probability 0.5 and send them away with probability 0.5. When we do send, we send them to a random location. We refer to this strategy as Random.

Another policy is simply sending an (almost) constant flow of containers on each edge. We used this policy also for initializing. We first determine the number of empty containers every city needs to send each day such that the expected inflow is equal to the expected outflow. If the net flow of a location is not an integer, lets say x, we send x containers with probability $\lceil x \rceil - x$ and $x + 1$ containers with probability $x - \lfloor x \rfloor$. We refer to this strategy as Balancing flow.

The next policy is the Rule of thumb. Here, we look three days back at the average number of orders in a certain city. We compare this with the number of available empty containers in that city. If there are more orders on average than there are available containers, that city wants to receive the difference. If there are more available containers than orders in a city, that city wants to send the difference away. We combine these requests to send/receive containers in a max flow min cost algorithm to determine the action we want to take.

The final policy we consider is the Rolling horizon [10]. Here we sample the orders 100 days ahead according to some arrival process. This arrival process is one of the environments we discussed. Then we solve an optimization problem where the objective is the total reward over those 100 days. The variables are the actions we can take each one of those 100 days. Then we solve this and we take the action of the first day. We then end up in a new state and sample again 100 days in the future and use this to make a new action.

Based on this principle we create another benchmark. However, unlike all the benchmark policies we discussed so far, the next benchmark is not a policy. We sample all the orders for one episode and we then determine the best actions for all the days at once. Given the orders, this is the best possible combination of actions one can take. We call this benchmark the hindsight model. The hindsight model cannot be used as a policy since we assume we know all the future orders at the present time, while this is obviously not the case for the other policies we discussed.

4.0.4 KPIs

During the evaluation of the different policies, we keep track of different KPIs (Key Performance Indicators). These KPIs give insight into how the policies behaves in the environment and the instance. The KPIs we use are:

- Total average reward
- Reward decomposition
- Utilization rate
- Fraction of orders that are fulfilled (per link)
- Fraction of movements that are done with empty containers (per link)
- Average state
- Number of containers that are not used in each city

The total average reward equals the sum of the rewards of the different cities averaged over the different days in each episode. The reward decomposition looks at which part comes from full travels, i.e. orders that we fulfill, and at the part that comes from the actions we take, i.e. the empty travels. This KPI might be interesting when two policies result in the same rewards. One might conclude that there are no large differences between the policies. However, when one policy sends many more empty containers away, this is reflected in the reward decomposition. This distinction would then favor one policy over the other.

The utilization rate equals the average usage of a container over time. We choose our instance in such a way that the utilization in the hindsight model lies around 80%. We still expect differences in utilization rates between policies. We also split the utilization rate in two parts: empty travels and full travels. Here we average over all the days and see how many containers are moving while empty and while being full respectively.

The fraction of orders that are fulfilled simply equals the number of orders that are fulfilled, divided by the total number of orders. When we look at the fraction of movements that are done with empty containers, we simply count the empty travels (per link) and divide this by the total number of travels (per link). This gives insight into the actions the policy tends to take.

The average state is the average state matrix of one episode, where the average is taken over all the days. Looking only at the first column, we only look at the first column, we see the number of containers that are not used in each city.

4.1 Results

We first look at the smaller instance with 10 cities in the (normal) Poisson environment. Here we see the different policies in detail. Then we consider the large instance. We also consider different environments to test the robustness. We still benchmark the Reinforcement Learning policy with other policies. Finally, we try to gain an insight in the reinforcement policy. We analyse which aspects are important in the decision making process of the policy.

4.1.1 Small instance

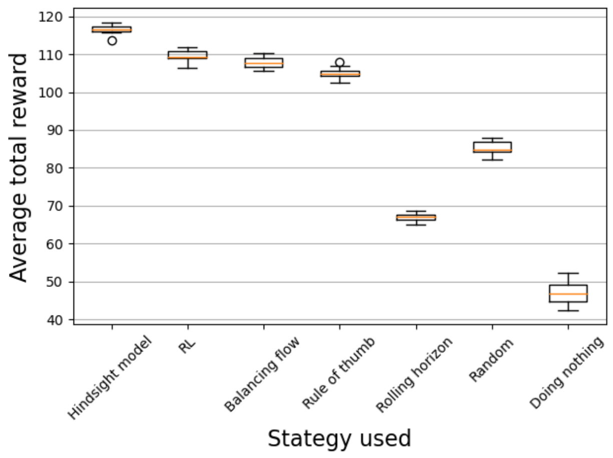

Figure 5: The daily average total reward of all the cities in 10 episodes over 100 runs.

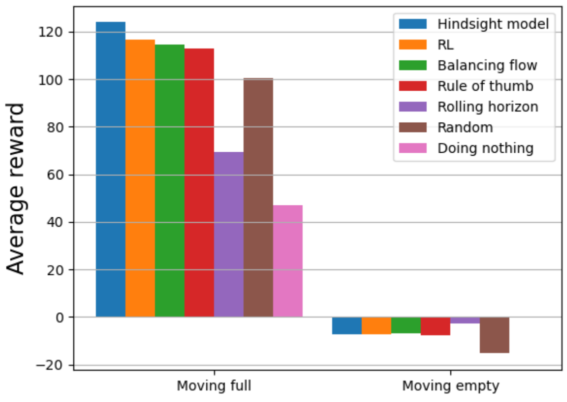

Figure 6: The average total reward decomposition.

Figure 5, shows the total reward from each policy. We see that the hindsight model has the highest average total reward. This is as expected, since this model takes optimal actions with knowledge of the future.

The RL policy has the second highest average total reward, this is only slightly more than Balancing flow and Rule of thumb. These differences are significant when we use a t-test for difference in means. We use a significance level of 0.05. It is surprising is that Random performs better than Rolling horizon. This is because there are naturally, most empty stationary containers at the center cities (city 4 and 5). At these cities, the fewest orders arrive, so every other city can make a better use of the containers. Doing nothing performs the worst, which is not surprising.

In Figure 6 we see the reward decomposition. We first compare the reward decomposition of RL, Balancing flow and Rule of thumb. We see that the difference in average total reward is mainly explained by the positive rewards from fulfilling orders, since the cost of moving empty containers is almost equal. This suggests that the RL policy sends the empty containers the most efficiently. We see that Rolling horizon has an extremely low cost from moving empty containers. This in turn suggests that Rolling horizon is too cautious with sending empty containers. This can be explained by the fact that the Rolling horizon policy fixes a strategy for 100 days and then takes only the first action. In an optimization stage used to determine one action, the algorithm might have decided to move more empty containers at a later stadium. However, since we start this optimization process again for each action, it might be the case that we postpone sending empty container every time. As a result we might never actually send these containers.

Figure 7 shows the fraction of orders that each policy fulfills. Here we have the same ordering as the average total reward. This is a logical consequence, since these two KPIs are closely correlated.

Figure 8 shows the utilization of the containers for each strategy. It is logical that the RL strategy has a higher full utilization and a lower empty utilization than Balancing flow and Rule of thumb. Since this implies the ordering in reward sizes we just discussed. It is interesting is that the total utilization using the RL strategy is also higher than the total utilization of Rule of thumb and Balancing flow. This again suggests that the RL policy uses its containers the most efficiently.

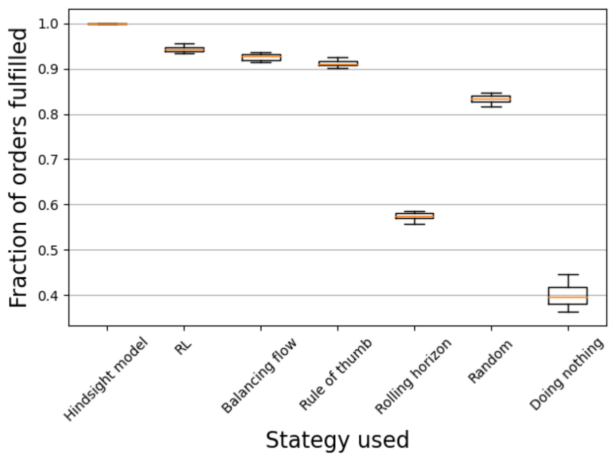

Figure 7: The fraction of orders fulfilled using different policies.

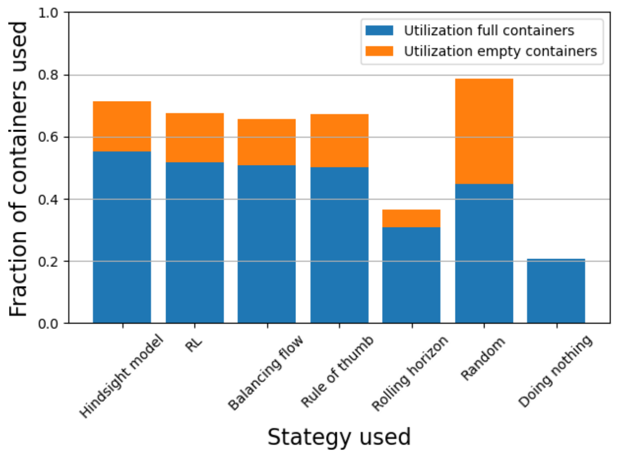

Figure 8: The utilization rate using different strategies.

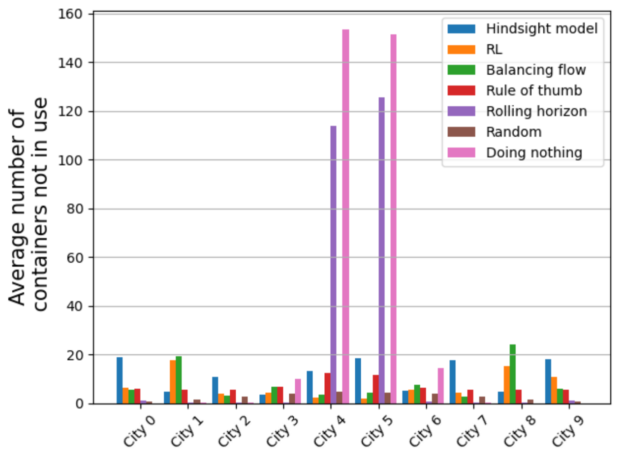

Figure 9: The number of empty containers staying per city using different strategies.

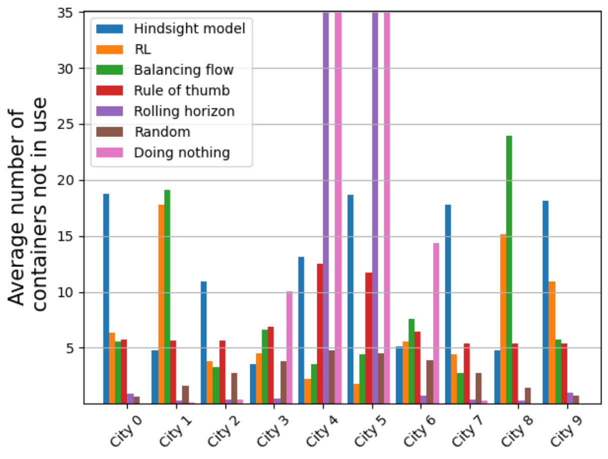

Figure 10: The number of empty containers staying per city using different strategies zoomed in.

Figure 9 and 10 show the average number of containers that remain unused after each day in every city for different policies. We see that the hindsight model lets most containers stay in the two outer cities (city 0 and city 9) when compared to the RL, Balancing flow and Rule of thumb. This has as an effect that the hindsight model can more easily accept requests from the outer cities. This is especially true for the Rule of thumb, as we will later see. Figure 11 shows the average state during one episode. We see that the RL strategy focuses on making sure that two cities (city 1 and 8) have enough containers to fulfill its orders. We see this also in the fact that city 1 and 8 have most of the stationary containers while using the RL strategy. We see that the Rule of thumb focuses more on the center cities (city 4 and 5). Balancing flow also focuses on city 1 and 8, however this seems to go at the expense of other cities.

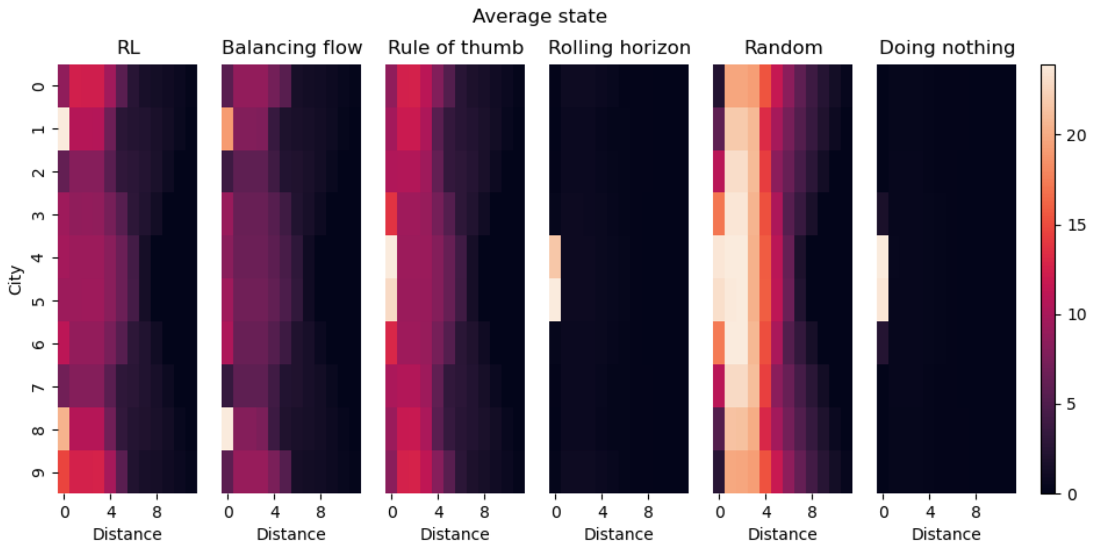

Figure 11: The average state in the simulation using different policies. Here the average is taken over both runs and episodes.

Figure 12 shows the fraction of orders fulfilled per link. Here we see again that the RL strategy focuses on city 8 and that this works, since almost all orders of city 8 are fulfilled. The same applies for Balancing flow. However, as noticed earlier, Balancing flows seems to focus too much on city 1 and 8. As a result the cities around 1 and 8 (city 0, 2, 7 and 9) have lower fractions of orders fulfilled.

We see that RL and Balancing flow fulfill slightly fewer orders in city 4 and 5 when compared to the Rule of thumb. However, this difference is very small and does not have big impact on the total reward, because the total number of orders is also small here.

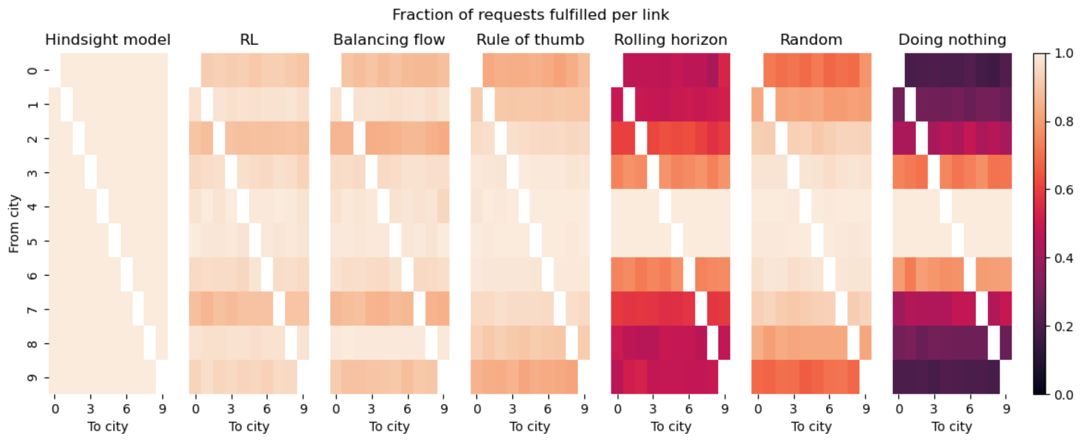

Figure 12: The fraction orders fulfilled per link using different policies.

Figure 13 shows the average fraction of movements that are empty per link. Figure 14 shows the total number of empty movements. We see that RL and the Balancing flow almost look the same. The main difference is that Balancing flow uses link (4,0) and link (5,9) while RL also uses similar links (3,0), (4,1) and (6,9), (5,8) respectively. When we compare this to Rule of thumb we see that Rule of thumb uses more links to send empty containers. However, those links are used less than the links that RL of Balancing flow use.

When we look at all the ”complex” policies, we see that they all have a similar structure. Namely, sending containers from inside to outside. The differences are about how much each specific link is used.

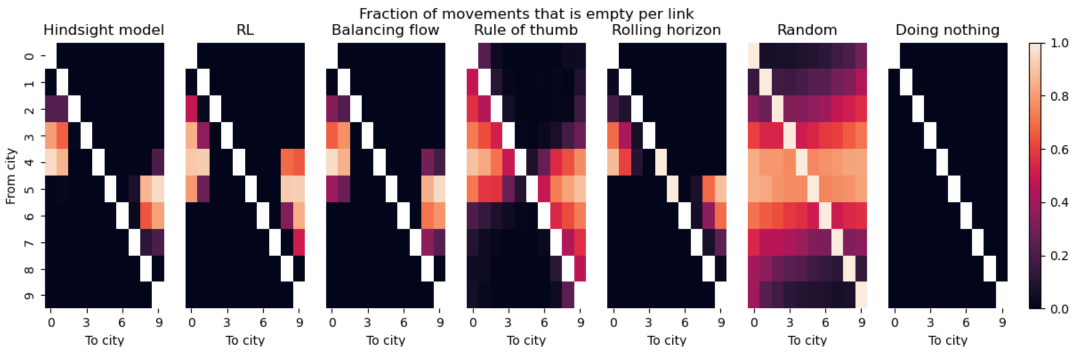

Figure 13: The fraction of movements that are empty per link using different strategies.

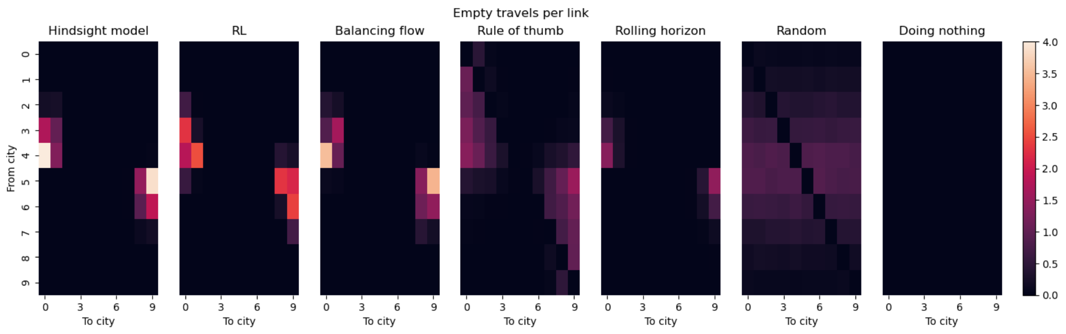

Figure 14: The number of empty movements per link using different policies.

4.1.2 Big instance

We now consider the larger instance based on data. This instance is more complex in how the states interact with each other. Also, there are simply more cities/countries we consider. Because of this, some visualizations are not useful. Especially the visualizations where we look at KPIs per link. For these KPIs we discuss only the most interesting ones. The others can be found in the appendix.

Figure 15 shows the average total reward per policy. The hindsight model again has the highest reward, which is a good sanity check. Compared to all other policies RL has the highest reward, the difference with Balancing flow, Rule of thumb and Rolling horizon are larger than in the small instance. Also the reward using Balancing flow is relatively much smaller, while the reward using Rolling horizon has relatively increased. This most likely has to do with the fact that the instance is much more complex and that there are many more containers. As a result, the state dependent policies (RL, Rule of thumb and Rolling horizon) perform better.

Figure 16 shows the positive rewards from fulfilling orders has the same ordering as the average total reward. In the negative reward, we see that RL and Rule of thumb approximately have the same costs. This means that RL uses the containers more efficiently. We can see that Rolling horizon again uses fewer empty travels. This can be explained by the same reasoning as before.

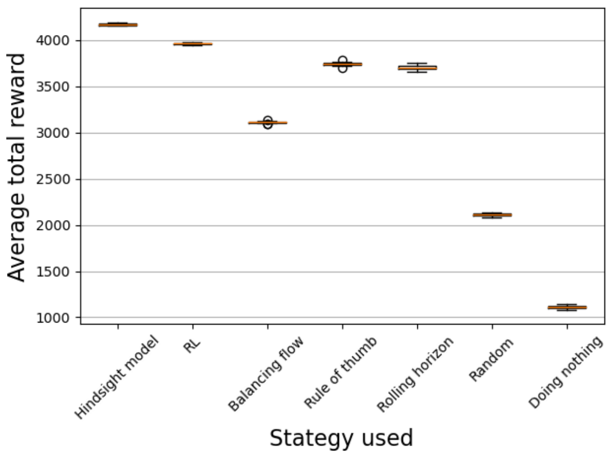

Figure 15: The daily average total reward of all the cities in 10 episodes over 100 runs.

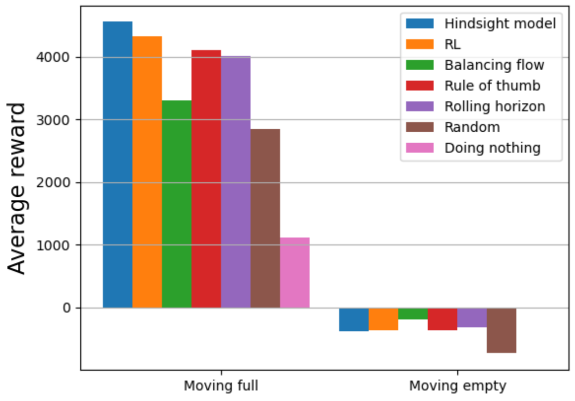

Figure 16: The reward decomposition.

In Figure 17 we see the same ordering as in Figure 15. This is again local and can be explained with the same reasoning as before. When a policy fulfills a large number of orders, the average rewards is likely also higher.

Figure 18 shows the utilization rate for each policy. Here we now see that RL has the highest utilization out of all the "smart" policies (Balancing flow, Rule of thumb and Rolling horizon). This has mainly to do with the fact that the full utilization is higher. Furthermore, we see that the utilization has the same shape as the reward decomposition would suggest.

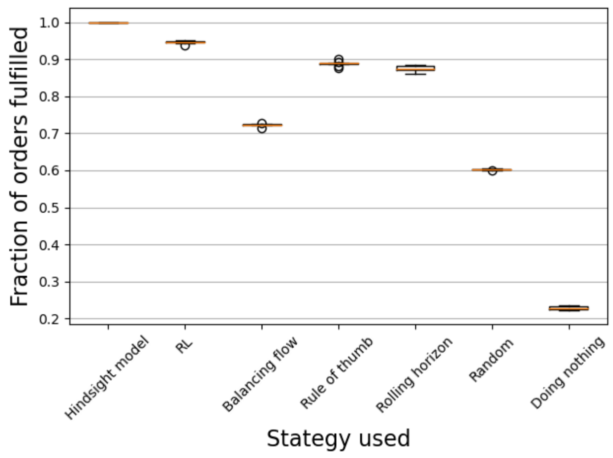

Figure 17: The fraction of orders fulfilled using different policies.

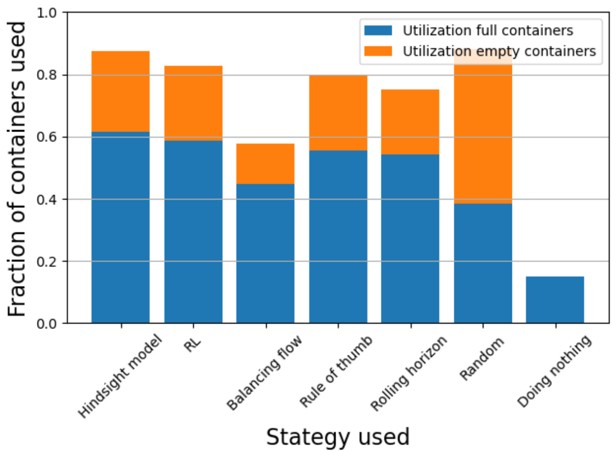

Figure 18: The utilization rate using different policies.

Next we looked at average states and KPIs per link. As stated before, this is now more difficult to visualize. For example, when we look at Figure 19 and compare it to Figure 4. In the input we see a large inflow to GBR, we see that the outflow of empty containers using RL is also large. When we compare this to the number of empty travels using other policies we do not see a large difference. Sometimes different links are used but the trend is the same. This is confirmed when we look at Figure 20. Here, we see that indeed the empty travels have the same structure in different strategies.

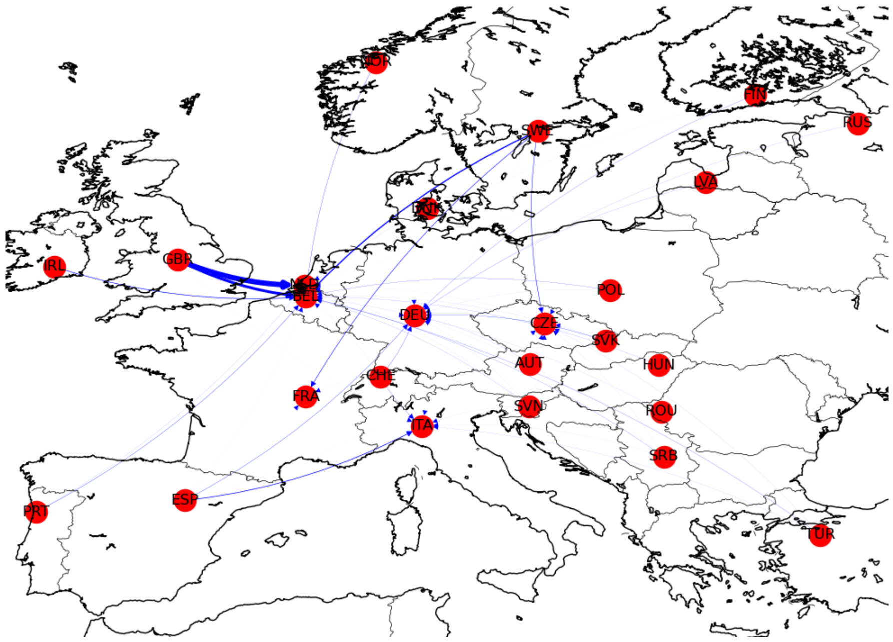

Figure 19: The empty travels per link using the RL policy. The size of the arrow denotes the number of containers.

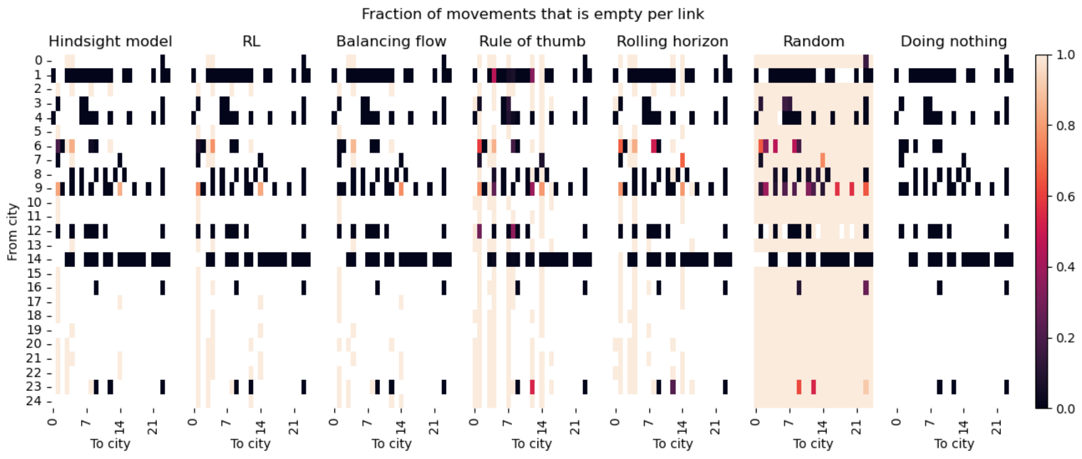

Figure 20: The fraction of empty travels per link using the different policies.

4.2 Robustness

We now look at the robustness of different strategies. We do this by testing policies in different environments. Some policies also depend on their training environment. We consider these policies in each combination of training environment and test environment. With the results we can determine what good candidates for training environments are. The policies that depend on a training environment are the RL policy and the Rolling horizon policy. We first look at all policies together and later we focus on RL specifically.

When we look at Table 2, we see that the influence of the test environment is much greater than the influence of the training environment. In most test environments we see the same qualitative ordering as we saw earlier when we evaluated in the Poisson environmet. The exception is the Moved Poisson with large permutations. We see that all RL policies and the Rule of thumb have similar average total rewards. In all other environments we see that the RL policies dominate all other policies. Therefore, we focus on more on the RL policies.

Policy		Test environment				
		Poisson	Negative Binomial	Moved Poisson small	Moved Poisson Large	Correlation model
Hindsight model		4169 (4.8)	4114 (36)	4176 (34)	4127 (80)	5231 (25)
RL	Poisson	3970 (1.0)	3651 (6.9)	3907 (6.5)	3550 (24)	4066 (1.9)
	Negative Binomial	3841 (1.1)	3553 (5.2)	3802 (5.8)	3685 (19)	3952 (1.9)
	Moved Poisson small	3969 (1.1)	3655 (6.3)	3911 (6.7)	3624 (21)	4068 (1.7)
	Moved Poisson Large	3877 (1.5)	3521 (6.7)	3827 (7.0)	3602 (18)	3988 (1.8)
	Correlation model	3880 (1.0)	3496 (7.3)	3827 (6.3)	3614 (21)	3992 (2.0)
Rolling horizon	Poisson	3711 (6.1)	3591 (14)	3707 (15)	3615 (54)	3774 (5.7)
	Negative Binomial	3738 (4.7)	3616 (14)	3704 (7.3)	3587 (40)	3813 (7.3)
	Moved Poisson small	3693 3.6)	3615 (12)	3703 (4.0)	3712 (58)	3796 (4.9)
	Moved Poisson Large	3712 (8.2)	3615 (16)	3703 (13)	3713 (6.4)	3799 (7.0)
	Correlation model	3706 (5.0)	3591 (15)	3701 (20)	3656 (13)	3781 (6.9)
Rule of thumb		3725 (3.7)	3347 (8.2)	3731 (10)	3706 (23)	3843 (3.2)
Balancing flow		3079 (6.0)	2889 (15)	3100 (16)	2939 (39)	3269 (7.2)
Random		2102 (3.9)	1747 (14)	2083 (17)	2121 (61)	2246 (7.4)
Doing nothing		1051 (2.3)	902 (10)	1050 (11)	968 (35)	1259 (9.6)

Table 2: The mean of the average total reward with inside the brackets the standard error.

Figure 21 shows the average total rewards for the different policies RL policies trained in the different environments. We run 100 simulations of 100 days, each simulation is one data point in the box plot. Here we see that the performance of the policies is not significantly different from each other in the environment Moved Poisson big. In all other environments, we see the same ordering of the performance of the policies. We see that training in the Poisson environment or in the Moved Poisson small environment are everywhere the two best policies. Therefore, training in environments where the variance is relatively small seems better for the performance.

When we train in an environment with a relative large variance, it is more likely that we end up in extreme states. Since these these states are significantly different from the more common state, the Q-function might not be accurate for these extreme states. The size of the adjustment we make to the parameters, depends on the difference between the predicted discounted reward (the Q-value) and the experienced discounted reward. Therefore, the changes to the parameters we make is larger in these extreme states. As a result, we might be overfitting on these states. This in turns results in a policy, which might mot be as effective on the more common states.

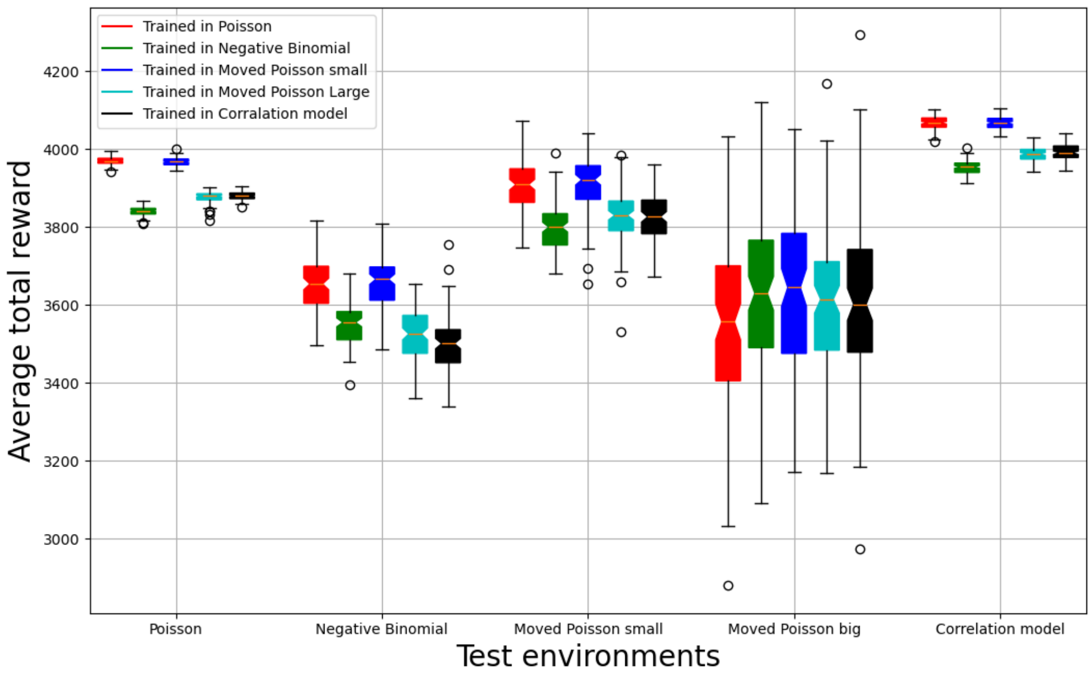

Figure 21: The daily average total reward for each RL policy in different test environments.

4.3 Importance

Finally, we look at the importance of other locations on the decision process of each location. To do this we look at which information is important for the value of the parameters for the parabola, i.e. the local Q-function. For this we use a random forest, we predict the parameters of the parabola, (b_0, b_3, b_2), based on the state matrix. Using this procedure we get for every location 3 random forests, so in total we have $24 \cdot 3 = 75$ random forests. In each random forest we use the importance feature of scikit-learn [11][12] to find the importance of each location for each location.

The importance is computed in the following way. At each node within the trees of the random forest, the optimal split is sought using the impurity. This is a computationally efficient approximation of the entropy. It measures how well a potential split is in this particular node. This means that the importance is dependent on the input data. As a result, when there is not much variability in a state, the importance is also most likely lower.

So, we have the importance of each location for each of the 3 parameters for each city. We take the average over the different locations for each parameter. The resulting value is the average importance of that location for other locations. We see these values in Figure 22. Here, we plot the average importance for each of these parameters against the scaled rate in, minus rate out. This is the average number of orders from a location minus the number of orders to a location divided by the standard deviation of that quantity, which we refer to as scaled net rate. The size of the marker is the sum of the rate in and the rate out, i. e. the total flow. Since size is often very difficult to quantify, we also divided the locations among 4 groups based on the total flow: the location with extremely small total flow, the locations with small total flow, the locations with average total flow and the location with large total flow.

We see in Figure 22 that all the locations with a high average importance have a scaled net rate of around zero. This can be explained by the fact that these locations are the most likely to have times where it has a shortage of empty containers and other times where it has too many empty containers. This means that sometimes other locations need to send containers to these locations. However, other times, when the locations with a scaled net rate around zero have too many empty containers, other locations can easily request more containers. This explains why these locations are more likely to be important to the decision making of other cities.

Another observation is that the locations with small total flow are more important than locations with large total flow. This is counter-intuitive. This is because, at these locations a lot of the total reward is earned. This would be explained by the fact that there are simply more containers in circulation in these locations. This intuition seems to be wrong. This might be because, the locations with a large total flow are more consistent. This is inherent to the Poisson distribution, since the standard deviation is the square root of the expected value. Another explanation might be the fact that, since the total flow is low that there are long periods of times when there are no orders at these locations. As a result, the empty containers accumulate.

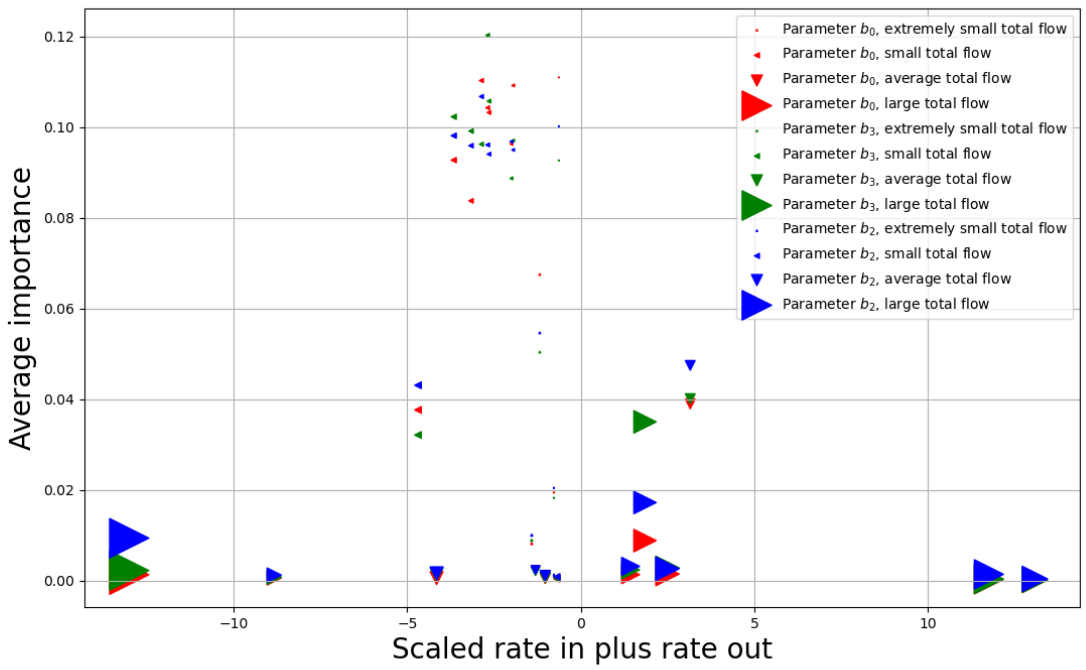

Figure 22: The average importance of different location in the decision making process.

5 Conclusion

In this report we discussed a way to create a Reinforcement Learning agent. We discussed the problems we faced and how we solved them. This gave us a policy for our logistic case. We evaluated this policy against other policies and in different instances and with different arrival distributions.

The first problem was the size of the state space. To solve this problem we used an episodic Semi-gradient algorithm. This algorithm is a parametric model instead of a model based on tabulation. The next problem was the dimensionality of the action space. We solved this by using local agents. This made the algorithm converge much faster and had a nice interpretation in the logistic case we used. Finally we solved some numeric errors by initializing and scaling the weights of the Q-functions.

Using the resulting algorithm we get a policy. The saw that this policy performs very well in each of the test instances. We saw that compared to other policies, the RL policy still performs well, espesially in the more complex instance. This shows us that the Reinforcement learning agent can generate a policy that performs well in a specific instance.

To test the robustness of the generated policies, we created 5 different arrival distributions. Then we generated policies in all 5 arrival distributions and tested them also on all 5 different arrival distributions. This way we can see the effect of the training arrival distribution and the effect of the test arrival distribution. We saw that the test arrival distribution had the greatest influence on the performance of the policies. We saw that policies with relative low variance in the training arrival distribution performed better in most environments. So, these policies were the most robust.

Additional, we looked at what the important information in the decision making process of the different agents was. Here we saw that locations with a net flow of around zero were import for the decision making process of other cities.

More generally, we saw how we can use Reinforcement Learning in a network setting. The use of local agents can be intuitive solution for a large action space for problems in a network setting.

6 Recommendations

In this report we created a RL agent and evaluated the resulting RL policy. We saw that this policy outperforms other more traditional policies in the instances and environments we used. Therefore, it is interesting to investigate the following:

- test the RL policy on even more complicated instances and environments,
- improve the benchmark policies for better comparison,

To improve the benchmark policies we can improve the Rolling horizon policy for a fairer comparison. We now used a simple version of a Rolling horizon policy, while we made a large number of improvement on the RL agent.

There are also options to improve the RL agent. For example, we now use a parabola for the local Q-functions. Another option would be to fit a function with more freedom. This might improve the accuracy of the Q-value in more extreme states. An example would be a neural network. The potential problem is then, that the number of parameters increases substantially. We can also increase the complexity of the Q-function, while still using parabola Q-functions. Currently, we estimate each coefficient of the parabola by using a linear combination of the state. This could be changed by introducing cross terms or square terms.

We can also make some changes earlier in the process. We chose to use local agents to reduce the complexity of the action space. Potentially, we can use a Wolpertinge architecture to try to solve this problem[13]. However, it is still not trivial how we would apply this to our problem. Applying this would come with its own disadvantages. For example, we would lose the interpretability of the local agents. We could now see one agent as a terminal manager, this is advantageous since it resembles the existing organizational structure.

References

[1] Richard Bellman and Robert Kalaba. Dynamic programming and statistical communication theory. *Proceedings of the National Academy of Sciences*, 43(8):749–751, 1957.

[2] Ronald A Howard. Dynamic programming and markov processes. 1960.

[3] Philippe Flajolet and Robert Sedgewick. *Analytic combinatorics*. cambridge University press, 2009.

[4] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[5] Oded Berger-Tal, Jonathan Nathan, Ehud Meron, and David Saltz. The exploration-exploitation dilemma: a multidisciplinary framework. *PloS one*, 9(4):e95693, 2014.

[6] Gabriel Alvarez. Can we make genetic algorithms work in high-dimensionality problems. *SEP–112*, pages 195–212, 2002.

[7] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[8] Luke Metz, Julian Ibarz, Navdeep Jaitly, and James Davidson. Discrete sequential prediction of continuous actions for deep rl. *arXiv preprint arXiv:1705.05035*, 2017.

[9] Richard H Jones and Francis Boadi-Boateng. Unequally spaced longitudinal data with ar (1) serial correlation. *Biometrics*, pages 161–175, 1991.

[10] Funda Sahin, Arunachalam Narayanan, and E Powell Robinson. Rolling horizon planning in supply chains: review, implications and directions for future research. *International Journal of Production Research*, 51(18):5413–5436, 2013.

[11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[12] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[13] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

7 Appendix

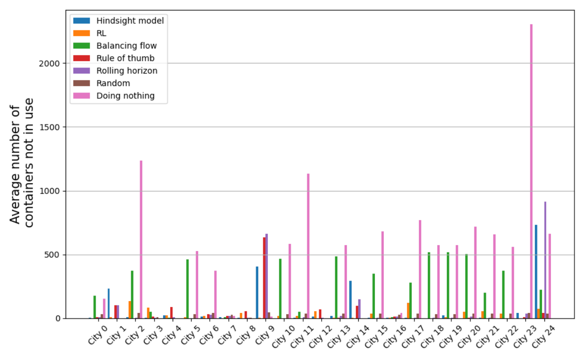

Figure 23: The number of empty containers staying per city using different strategies.

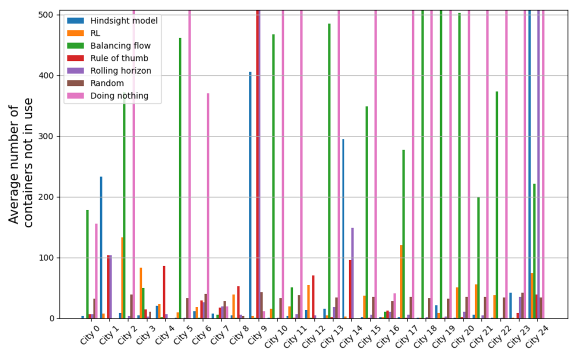

Figure 24: The number of empty containers staying per city using different strategies zoomed in.