
LINER SHIPPING NETWORK DESIGN WITH REINFORCEMENT LEARNING

Utsav Dutta, Yifan Lin, Zhaoyang Larry Jin

Data Science

C3 AI

Redwood City, California, US

{utsav.dutta, yifan.lin, larry.jin}@c3.ai

ABSTRACT

This paper proposes a novel reinforcement learning framework to address the Liner Shipping Network Design Problem (LSNDP), a challenging combinatorial optimization problem focused on designing cost-efficient maritime shipping routes. Traditional methods for solving the LSNDP typically involve decomposing the problem into sub-problems, such as network design and multi-commodity flow, which are then tackled using approximate heuristics or large neighborhood search (LNS) techniques. In contrast, our approach employs a model-free reinforcement learning algorithm on the network design, integrated with a heuristic-based multi-commodity flow solver, to produce competitive results on the publicly available LINERLIB benchmark. Additionally, our method also demonstrates generalization capabilities by producing competitive solutions on the benchmark instances after training on perturbed instances.

Keywords liner shipping · network design · neural network · reinforcement learning

1 Introduction

The liner shipping industry is the backbone of global maritime trade. It plays a critical role in the international supply chain, ensuring the efficient movement of merchandise across the globe. This industry involves the design and operation of container vessels that traverse fixed maritime routes to transport goods between ports efficiently and profitably. The strategic planning of these routes plays a pivotal role in optimizing both the revenue of ocean freight companies and the operational efficiency of their vessels. Well-designed shipping networks not only enhance profitability but also reduce the total number of vessels utilized, leading to lower maintenance costs and decreased emissions.

The Liner Shipping Network Design Problem (LSNDP) addresses this complex business challenge by modeling this as a mathematical optimization problem. The goal of the LSNDP is to design optimal vessel routes and allocate cargo flows across the network to maximize overall profitability. As a specialized routing problem, the LSNDP falls under the broader category of combinatorial optimization problems. Similar routing problems include the well-known Traveling Salesman Problem (TSP) and Vehicle Routing Problem (VRP). Similar to these, the LSNDP is classified as NP-hard, due to which solving large-scale instances to optimality with traditional OR approaches such as Mixed-Integer Programming (MIP) is computationally intractable.

However, recent advances in deep learning for routing problems have introduced an alternative approach to solving NP-hard combinatorial optimization problems. Kool et al. [2018] applied Reinforcement Learning (RL) to the Traveling Salesman Problem (TSP) and several variants of the Vehicle Routing Problem (VRP), demonstrating the potential of RL in this domain. Building on this, Joshi et al. [2019] enhanced the RL framework by incorporating Graph Convolutional Networks (GCN), which yielded promising results for TSP. These learning-based approaches achieved results comparable to traditional OR methods in terms of optimality, while also demonstrating impressive generalizability. This opens up avenues to learn general purpose policies from a diverse dataset and use these to infer high quality solutions on new unseen data points.

In this paper, we present a learning-based approach to the LSNDP. We break down the LSNDP into two sub-problems, as is classically done in OR-based approaches to this problem, namely the Network Design Problem (NDP) and the Multi Commodity Flow Problem (MCF). We formulate the NDP as a Markov Decision Process (MDP). By integrating a reinforcement learning (RL) algorithm with a heuristic-based multi-commodity flow solver, our method achieves competitive results on the publicly available LINERLIB¹ benchmark. Our approach offers its value in two distinct ways. Firstly, our approach demonstrates that it can serve as a competitive end to end optimizer, in a similar vein to traditional OR solvers, and secondly, our approach shows signs of generalization capabilities by learning a robust policy that can provide high quality solutions for perturbed instances.

The rest of the paper is organized as follows. We introduce the Liner Shipping Network Design Problem (LSNDP) and its decomposed sub-problems: multi-commodity flow (MCF) and network design problem (NDP) in Section 3. For the decomposed network design problem, we introduce our reinforcement learning framework in Section 4. We demonstrate the quality and generalizability of the proposed approach on the LINERLIB benchmark in Section 6. Finally, we conclude the paper in Section 7.

2 Related Work

The Liner Shipping Network Design Problem (LSNDP) has been extensively researched within the operations research (OR) community for several decades. To support benchmarking efforts for the LSNDP, Brouer et al. [2014] introduced a standardized dataset known as LINERLIB¹, which includes seven real-world instances of the problem, each varying in scale. In a comprehensive review of the LSNDP literature, Christiansen et al. [2020] discussed the standardized formulation of the problem widely accepted by the OR community and reviewed the development of the OR-based approaches typically used in this domain. The performance of leading algorithms is benchmarked on the LINERLIB dataset in their work. According to the authors, traditional OR approaches to the LSNDP can generally be categorized into the following main types:

- Holistic MIP-based formulations, which address both the service design and the multi-commodity flow aspects simultaneously, as exemplified by the work of Plum et al. [2014] and Wang and Meng [2014].
- Local search-based methods, which explore variations from a predefined set of candidate services, such as the approach described by Meng and Wang [2011] and Balakrishnan and Karsten [2017].
- Two-stage algorithms: These decompose the problem into two distinct phases: first solving the network design problem (NDP) and then the multi-commodity flow (MCF) problem separately. A common method, as used by Brouer et al. [2014] and Thun et al. [2017], involves designing the services (i.e., NDP) first and subsequently routing the containers through the designed network (i.e., MCF). Alternatively, Krosgaard et al. [2018] propose a reverse approach, where containers are first flowed through a relaxed network before finalizing the network design based on the flow. This class of approaches has generally proven to be the most effective, yielding reasonable solutions on the largest instance in the LINERLIB dataset, where other methods have failed.

Despite these advancements, two significant challenges persist across all OR methods. First, scalability remains a critical issue due to the NP-hard nature of the LSNDP, with large, real-world instances still unsolved. Second, generalizability is a major limitation, as even small perturbations in problem instances often necessitate a complete reconstruction of the solution, requiring a similar level of computational effort as the original problem.

In the past decade, Reinforcement Learning (RL) has gained increasing attention as a method for solving combinatorial optimization problems. Khalil et al. [2017] pioneered the use of RL to tackle the Maximum Cut and Minimum Vertex Cover problems, combining graph embeddings with Q-learning to generate heuristics. Hu et al. [2017] made the first attempt at applying RL to the Bin Packing Problem. Within the domain of routing problems, Vinyals et al. [2015] introduced the Pointer Network (PN) to address the Traveling Salesman Problem (TSP), employing attention mechanisms to map inputs to outputs. Bello et al. [2016] built on this work by applying the Actor-Critic algorithm to improve PN performance. Nazari et al. [2018] extended the RL approach to the Vehicle Routing Problem (VRP), enhancing the Pointer Network by replacing the Long-Short Term Memory (LSTM) encoder with a 1-D convolutional embedding.

More recent advances in RL for routing problems include the work of Kool et al. [2018], who developed a construction-heuristic learning approach applied to the TSP, VRP, and other related routing challenges. Their approach enhanced the encoder by introducing a Transformer-like attention mechanism, while the decoder maintained a similar structure to the original PN. Joshi et al. [2019] further advanced the field by incorporating a Graph Convolutional Network (GCN)

¹<https://github.com/blof/LINERLIB>

for the encoder, alongside a highly parallelized, non-autoregressive beam search roll-out. This method outperformed autoregressive models in terms of solution quality for TSP. Building on these developments, Fellek et al. [2023] introduced a more sophisticated graph embedding scheme, leveraging a multi-head attention structure that embeds edge information. Their approach demonstrated notable improvements for VRP, further advancing the effectiveness of RL in solving complex routing problems.

Significant strides have been made to improve the scalability and generalizability of RL-based approaches. Drori et al. [2020] tackled edge-selection problems by converting them into node-selection tasks using line graphs and applied Graph Attention Networks (GAT) for embedding. Their method was tested on various NP-hard combinatorial optimization problems, including TSP, and showed that inference time scaled linearly with problem size. Other key advancements include the Adaptive Multi-Distribution Knowledge Distillation (AMDKD) framework proposed by Bi et al. [2022], and the work of Fu et al. [2021], which employed Monte Carlo Tree Search (MCTS) alongside a heat map generated from a pre-trained supervised learning model to effectively scale solutions for arbitrarily large VRP instances.

The advancements of the above-mentioned RL methods show potential for application to a wider range of routing problems beyond TSP and VRP, including the Liner Shipping Network Design Problem (LSNDP). This paper attempts to take strides in that direction, by applying RL to a richer, and harder class of NP-hard combinatorial optimization problems, which has significant applications to real-world applications.

3 LSNDP

Brouer et al. [2014] and Christiansen et al. [2020] both provided a comprehensive definition of the Liner Shipping Network Design Problem (LSNDP): Given a set of ports, a fleet of container vessels, and a collection of demands specified in the quantity of Forty-Foot Equivalent units (FFE) with designated origins, destinations, and shipping rates, the objective is to design a set of cyclic sailing routes for the vessels (i.e., services) that maximize revenue from fulfilling the demands while minimizing the overall operational costs of those vessels.

It is important to note that, unlike the VRP or other dispatch problems, the LSNDP does not focus on the specific scheduling of vessels. Instead, it generates a set of services, each representing a round-trip route with a fixed itinerary of ports, called at regular intervals, typically at a weekly or bi-weekly frequency. Vessel assignments are subsequently determined to support these services. For example, if a round-trip route takes six weeks to complete, six vessels will be needed to maintain a weekly service. Additionally, the demand in LSNDP is normalized according to these service frequencies, simplifying the problem formulation.

Services in the LSNDP are categorized based on the topological structure of their routes. A simple service follows a round-trip route where vessels visit each port exactly once, forming a single circular loop. However, services are often non-simple, meaning that some ports are visited more than once along the same route. Visually, these non-simple services form multiple loops. Depending on their structure, they can be classified as butterfly services, pendulum services, or complex services. Figure 1 illustrates an example of a butterfly service generated in the LSNDP, where ports are labeled using their corresponding UNLOCODE². Note that one hub port London (GBLON) is visited twice.

Building on the basic definition provided above, several variations of the LSNDP have been extensively studied in the literature. These variations introduce additional factors such as transit time, which imposes time constraints on demand, transshipment costs, vessel speed optimization, and penalties for leaving a portion of the demand unsatisfied (rejected demand). In line with most studies that benchmark their results using the LINERLIB dataset, this work focuses on the LSNDP variation that incorporates transshipment costs and rejected demand, while excluding considerations for transit time and vessel speed optimization. Additionally, we assume that vessels operate strictly at their designed speed and permit fractional vessel assignments, which simplifies the modeling of vessel deployment and optimizes resource allocation.

As discussed in Section 2, a widely adopted approach within the traditional operations research (OR) community for solving the Liner Shipping Network Design Problem (LSNDP) is to decompose it into two closely related sub-problems: the multi-commodity flow (MCF) problem and the network design problem (NDP). Detailed descriptions of the MCF and NDP can be found in Appendix A.

In our proposed reinforcement learning (RL) approach, we aim to leverage this two-tier framework. Rather than formulating the NDP as a Mixed-Integer Problem (MIP), we represent it as a Markov Decision Process (MDP), where round-trip services are generated sequentially. At each step t , the generation of a complete service is treated as an action A_t , with the step index t indicating the number of services generated up to that stage.

²<https://unece.org/trade/uncefact/unlocode>



Figure 1: Example of a butterfly service with a hub port at London (GBLON).

The MCF serves as a key component in the reward evaluation function at each step, defined by:

$$R_{t+1} = \eta_{t+1} - \eta_t, \quad (1)$$

where R_{t+1} is the reward in the MDP context, and η_{t+1} and η_t represent the profit of the network with all services generated up to steps $t + 1$ and t , respectively. For further details on the MDP formulation, refer to Appendix C.

To support the RL approach, an advanced MCF algorithm is crucial for enabling fast reward evaluation, as the reward function is invoked hundreds of millions of times during the training process. Additionally, the quality of the NDP solution is partly influenced by the performance of the MCF algorithm since variations in the reward signals can steer the RL training in different directions. In this study, we have implemented a basic version of the MCF algorithm (see Appendix B), which is less sophisticated than the state-of-the-art heuristic implementations, such as those by Krogsgaard et al. [2018], and also lags behind MIP-based solutions.

Despite the limitations of our MCF implementation, we were still able to develop an RL-based solution for the NDP component of the LSNDP. In the following section, we introduce two approaches for finding optimal NDP solutions by parameterizing the policy as neural networks.

4 Policy Neural Network Design

In this section, we present two modeling approaches for parameterizing the policy π_θ as neural networks: the encoder-only approach and the encoder-decoder approach. After representing the NDP as a Markov Decision Process (MDP), either approach can produce a parameterized policy π_θ that guides the actions at each step t . This process can be described by the following sampling equation:

$$A_t \sim \pi_\theta(\cdot | S_t), \quad (2)$$

where the action A_t in the context of the NDP comprises of two components: the vessel selection $A_{v,t} \in \mathbb{R}^V$, which determines the vessel type to be deployed from the overall fleet, and the service selection $A_{p,t} \in \mathbb{R}^P$, which specifies an ordered sequence of ports:

$$A_t = [A_{v,t}, A_{p,t}]. \quad (3)$$

The state at step t is represented by two components: $S_t = \{S_{t,g}, S_{t,v}\}$. Here, $S_{t,g}$ captures the state of the shipping network as a graph, and $S_{t,v}$ describes the status of the available vessels. These components are defined as follows:

$$S_{t,g} = \{\mathbf{f}_p, \mathbf{f}_e\}, \quad (4)$$

$$S_{t,v} = \mathbf{v}_t \in \mathbb{R}^{V \times D_v}. \quad (5)$$

In this representation, $\mathbf{f}_p \in \mathbb{R}^{(P+1) \times 2}$ contains the two-dimensional features of all nodes in the graph, where each node corresponds to a port in the shipping network. The notation P represents the total number of ports in a given problem instance, while the two-dimensional features capture the total incoming and outgoing demand for each port at step t . An additional node is included to track the global features of the graph, resulting in a total of $P + 1$ nodes. The feature matrix $\mathbf{f}_e \in \mathbb{R}^{E \times D_e}$ describes the edges in the graph, where each edge represents a potential leg in a service connecting two ports. Here, E represents the total number of possible port pairs that a vessel can traverse in a single leg, and D_e is the dimension of the edge features, with $D_e = 6 + |S|$, where $|S|$ is the maximum number of services allowed in the problem instance. Note that features \mathbf{f}_e and \mathbf{f}_p are step t dependent, here we drop the subscript for simplicity.

The notation \mathbf{v}_t represents the feature set for each vessel class, including the number of vessels remaining to be deployed in each class. Here, V denotes the total number of vessel classes in the problem instance, and D_v represents the dimension of the vessel features. For more details on the state representations, please refer to Appendix C.

Both the encoder-only and encoder-decoder approaches follow a similar sequence: selecting the vessel class first, and then determining the service rotation. The sequence repeats at each step t until one of the following conditions is met: the maximum number of services $|S|$ is reached, all demands are satisfied, or all vessels are exhausted. The key difference lies in how the service rotation $A_{p,t}$ is generated. Note, $A_{p,t}$ represents the selection of one service for the shipping network, which involves determining a sequence of ports to include in the service. This can be approached in two ways: either selecting all ports simultaneously and deciding their sequence afterward (encoder-only approach) or selecting each port sequentially (encoder-decoder approach).

The encoder-only approach uses a one-shot rollout, where a complete sequence of actions is generated in a single step, based on a policy that predicts the entire sequence simultaneously. In contrast, the encoder-decoder approach relies on an autoregressive rollout, where the policy generates ports sequentially one at a time, using each previously selected port as input for the next selection. The complete sequence of ports once generated constitutes an action.

4.1 Encoder-Only Approach with One-Shot Rollout

Figure 2 demonstrates the workflow of the encoder-only approach. Instead of defining vessel selection as sampling from a stochastic policy, we use a simple deterministic heuristic for vessel selection. Specifically, we select the “largest available vessel” as the action $A_{v,t}$, where “largest” refers to the vessel class with the highest capacity, measured in FFEs:

$$A_{v,t} = \underset{\substack{v \in [1, V] \\ (\mathbf{v}_t)_{v,1} > 0}}{\operatorname{argmax}} \operatorname{Capacity}(v). \quad (6)$$

Here, v represents the index of the vector \mathbf{v}_t , and $\operatorname{Capacity}(v)$ denotes the capacity of the vessel class corresponding to index v . The constraint $(\mathbf{v}_t)_{v,1} > 0$ ensures that only vessel classes with available vessels are considered for selection.

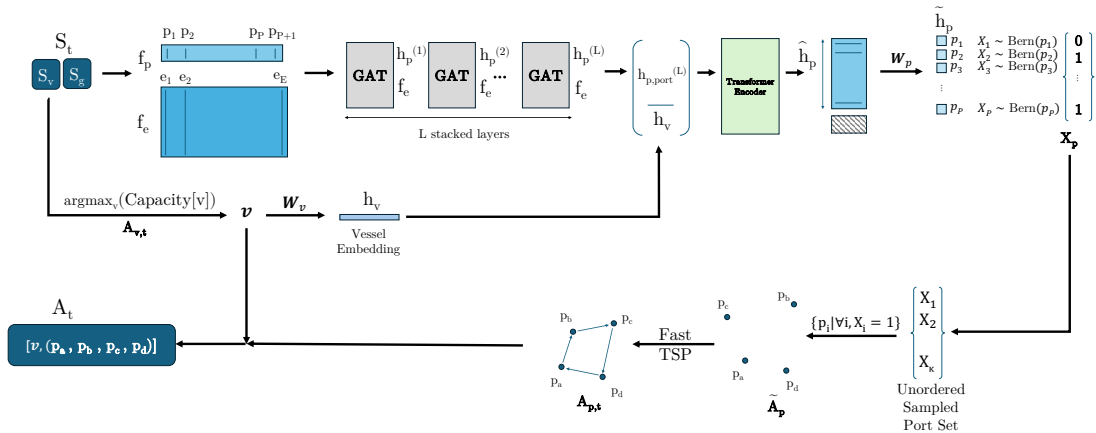


Figure 2: Encoder policy diagram for NDP.

Our encoder is composed of L sequential layers of Graph Attention Networks (GAT) with an embedding dimension of H , followed by a standard Transformer encoding layer. The GAT layers learn contextual representations for each port by leveraging the underlying graph structure, while the Transformer encoder enables attention across all port pairs,

capturing interactions between them. We omitted the positional embedding layer, as our embedding structure does not have inherent temporal properties.

We pass the graph representation, \mathbf{f}_p and \mathbf{f}_e , through the GAT layers. After processing it through L graph attention layers, we obtain a dense, contextualized representation $\mathbf{h}_p^{(L)} \in \mathbb{R}^{(P+1) \times H}$. Mathematically, this process can be expressed as follows:

$$\mathbf{h}_p^{(1)} = \mathbf{GAT}^{(1)}(\mathbf{f}_p, \mathbf{f}_e) \in \mathbb{R}^{(P+1) \times H}, \quad (7)$$

$$\mathbf{h}_p^{(l)} = \mathbf{GAT}^{(l)}(\mathbf{h}_p^{(l-1)}, \mathbf{f}_e) \in \mathbb{R}^{(P+1) \times H}, \quad (8)$$

where the superscript (l) denotes the l -th layer of the network. Note that only the node features are transformed into the dense representation $\mathbf{h}_p^{(l)}$, while the edge features \mathbf{f}_e remain unchanged. Through this iterative message-passing process, information from both the nodes and edges is effectively aggregated into the node representations.

It is important to note that the output from the GAT consists of two components: the port embeddings and the global embedding:

$$\mathbf{h}_p^{(L)} = [\mathbf{h}_{p,\text{port}}^{(L)}, \mathbf{h}_{p,\text{global}}^{(L)}] \in \mathbb{R}^{(P+1) \times H}, \quad (9)$$

where $\mathbf{h}_{p,\text{port}}^{(L)} \in \mathbb{R}^{P \times H}$ represents the embeddings of the ports, and $\mathbf{h}_{p,\text{global}}^{(L)} \in \mathbb{R}^{1 \times H}$ corresponds to a global node embedding that serves as a general representation of the entire graph. In the encoder-only approach, only the port embeddings are utilized, while the global embedding is used by the decoder.

With the port information fully encoded, we now shift our focus to the vessel information. Based on the vessel action $A_{v,t}$ described earlier, v denotes the index of the selected vessel class. In this step, only the features of the selected vessel class, $(\mathbf{v}_t)_v \in \mathbb{R}^{D_v}$, are used in the workflow. The vessel state is then encoded using a matrix multiplication:

$$\mathbf{h}_v = \mathbf{W}_v (\mathbf{v}_t)_v \in \mathbb{R}^H, \quad (10)$$

where $\mathbf{W}_v \in \mathbb{R}^{H \times D_v}$ is a linear transformation that maps the vessel state features to a dense representation of dimension H .

After encoding both the graph state and vessel state into dense matrices, they are processed through a standard Transformer encoder. The resulting updated graph embedding $\hat{\mathbf{h}}_p$ is then passed through a Sigmoid function:

$$[\hat{\mathbf{h}}_p, \hat{\mathbf{h}}_v] = \mathbf{Transformer}(\mathbf{h}_{p,\text{port}}^{(L)}, \mathbf{h}_v) \in \mathbb{R}^{(P+1) \times H}, \quad (11)$$

$$\tilde{\mathbf{h}}_p = \frac{1}{1 + e^{-(\mathbf{W}_p \hat{\mathbf{h}}_p)^T}} \in \mathbb{R}^P, \quad (12)$$

where \mathbf{W}_p is a linear transformation that maps the graph embedding to a P -dimensional vector. The Sigmoid function produces $\tilde{\mathbf{h}}_p \in \mathbb{R}^P$, with all elements constrained within the range $[0, 1]$. The resulting embedding is then subjected to a Bernoulli sampling process, defined by:

$$\mathbf{X}_p = \mathbf{Bernoulli}(\tilde{\mathbf{h}}_p) \in \{0, 1\}^P. \quad (13)$$

We define the set $\tilde{A}_p = \{i | (\mathbf{X}_p)_i = 1, \forall i \in \mathbf{X}_p\}$, which represents an unordered set of ports to be included in a service. A port is included in the service if the corresponding value in \mathbf{X}_p is 1, and excluded if the value is 0.

To generate an ordered set representing a service, we use a fast approximate TSP solver (see [Shintyakov \[2017\]](#)) to transform the unordered set into an ordered rotation:

$$A_{p,t} = \mathbf{TSP}(\tilde{A}_p, \mathbf{f}_e). \quad (14)$$

The TSP solver requires additional static graph features from \mathbf{f}_e (such as the distance matrix between ports) to determine the optimal port-call sequence. The resulting ordered set of ports is treated as the service selection action for the current step, $A_{p,t}$. Here, we include the time step subscript t to maintain consistency with the notation used in other sections. Assembling $A_{v,t}$ from Eq. 6 and $A_{p,t}$ from Eq. 14, action A_t is completed as defined in Eq. 3.

4.2 Encoder-Decoder Approach with Autoregressive Rollout

Equations 11 to 14 describe the one-shot rollout for the encoder-only approach, where the probability of each port being included in a service is modeled independently within each action A_t . While this method is straightforward and intuitive,

it treats the inclusion of each port as independent, limiting its ability to account for dependencies between selected ports. In contrast, the encoder-decoder approach with autoregressive rollout, explicitly models these dependencies, where the decision to include an additional port depends on the previously selected ports in the current and all previous services.

In the autoregressive rollout, the action A_t is generated sequentially, involving multiple sub-steps within a single step t . The process begins with a sub-step, denoted as τ , for vessel selection, followed by several sub-steps to select ports, thereby completing the generation of a single service. It is important to note that in this approach, vessel selection is also determined by the policy π_θ , which is parameterized by a neural network, rather than the rule-based selection used in the encoder-only approach (Eq. 6).

The embeddings used for the autoregressive rollout are generated as outputs from the encoder phase. Specifically, for the port embeddings, we define:

$$\check{\mathbf{h}}_p = \mathbf{Transformer}(\mathbf{h}_{p,\text{port}}^{(L)}) \in \mathbb{R}^{P \times H}, \quad (15)$$

where $\mathbf{h}_{p,\text{port}}^{(L)} \in \mathbb{R}^{P \times H}$ is the port embedding previously defined in Eq. 9. For the vessel embeddings, we similarly define:

$$\check{\mathbf{h}}_v = \mathbf{W}'_v \mathbf{v}_t \in \mathbb{R}^{V \times H}. \quad (16)$$

where $\check{\mathbf{h}}_v$ represents the embeddings for all vessel classes, rather than just the selected vessel class as used in Eq. 10 for the encoder-only approach. Note the $\mathbf{W}'_v \in \mathbb{R}^{H \times D_v}$ is distinct from \mathbf{W}_v . Next, we define the overall embedding for the decoder:

$$\mathbf{h}_{\text{embed}} = \begin{bmatrix} \check{\mathbf{h}}_p \\ \check{\mathbf{h}}_v \\ \mathbf{h}_{\text{BOS}} \end{bmatrix} \in \mathbb{R}^{\bar{N} \times H}, \quad (17)$$

where $\mathbf{h}_{\text{BOS}} \in \mathbb{R}^H$ represents the embedding for the ‘‘beginning of service’’ (BOS). This vector is randomly initialized and remains static throughout the entire service generation process. The notation $\bar{N} = P + V + 1$ reflects the total dimensionality of the embedding, where P is the number of ports, V is the number of vessel classes, and the additional 1 corresponds to the BOS. It’s important to note that the embeddings $\check{\mathbf{h}}_p$ and $\check{\mathbf{h}}_v$ vary with each step t , but remain constant across all sub-step τ ’s. For simplicity, we have omitted the t subscripts in this equation.

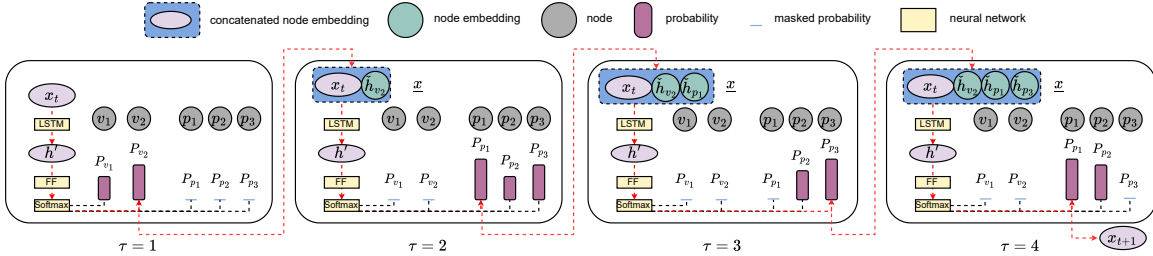


Figure 3: LSTM-based decoder for NDP. The decoder takes in \mathbf{x}_t generated from the previous steps and builds $\underline{\mathbf{x}}$ sequentially. The example shows how a full service $A_t = (v_2, p_1, p_3)$ is generated sequentially over $\tau = 1, 2, 3, 4$ within step t . Note that at $\tau = 4$, p_1 is selected again, which closes the circle and ends A_t .

Figure 3 illustrates how the agent progresses through each sub-step τ within a step t , using Long Short-Term Memory (LSTM) to guide the rollout process. This can be mathematically expressed as follows:

$$\mathbf{h}'_1, \mathbf{h}_1, \mathbf{c}_1 = \mathbf{LSTM}^{(1)}(\mathbf{x}_t, \mathbf{h}_0, \mathbf{c}_0), \quad (18)$$

$$\mathbf{h}'_\tau, \mathbf{h}_\tau, \mathbf{c}_\tau = \mathbf{LSTM}^{(\tau)}(\underline{\mathbf{x}}, \mathbf{h}_{\tau-1}, \mathbf{c}_{\tau-1}). \quad (19)$$

Here, $\mathbf{x}_t \in \mathbb{R}^{H \times n_0}$ represents the embeddings of all vessels and ports selected in prior services up to step t , while $\underline{\mathbf{x}} \in \mathbb{R}^{H \times n}$ extends this by including the embeddings of vessels and ports selected up to sub-step τ , where $n = n_0 + \tau$. The cell state of the LSTM at sub-step τ , denoted by \mathbf{c}_τ , is initialized as

$$\mathbf{c}_0 = \mathbf{h}_{p,\text{global}}^{(L)} \in \mathbb{R}^H, \quad (20)$$

where $\mathbf{h}_{p,\text{global}}^{(L)}$ is the global embedding defined in Eq. 9. The hidden state of the LSTM at sub-step τ , represented by \mathbf{h}_τ , is initialized as:

$$\mathbf{h}_0 = \frac{1}{P} \sum_{i=1}^P (\check{\mathbf{h}}_p)_i \in \mathbb{R}^H. \quad (21)$$

Once the output of the LSTM $\mathbf{h}'_\tau \in \mathbb{R}^{H \times n}$ is generated through Eq. 19, it is passed through a fully connected feed-forward (**FF**) layer with ReLU activation, which transforms the embedding from dimension H to \bar{N} . To enhance stability during training, layer normalization (**LN**) is applied to the resulting embeddings:

$$\hat{\mathbf{h}}^\tau = \text{LN}(\text{FF}(\mathbf{h}'_\tau)), \quad (22)$$

$$\hat{\mathbf{h}}^\tau = [\hat{\mathbf{h}}_1^\tau, \dots, \hat{\mathbf{h}}_n^\tau] \in \mathbb{R}^{\bar{N} \times n}. \quad (23)$$

The last embedding vector $\hat{\mathbf{h}}_n^\tau \in \mathbb{R}^{\bar{N}}$ is then processed through a Softmax layer to produce the final output probabilities:

$$\tilde{\mathbf{h}}_\tau = \frac{e^{\hat{\mathbf{h}}_n^\tau}}{\sum_{j=1}^{\bar{N}} e^{(\hat{\mathbf{h}}_n^\tau)_j}} \in \mathbb{R}^{\bar{N}}. \quad (24)$$

The probability distribution is filtered through a masking rule before it is used to sample the index of the vector which represents the next vessel or port in the service. The embedding of the selected vessel or port is then appended to $\underline{\mathbf{x}}$, which will serve as the input vector for the subsequent sub-step:

$$i \sim \mathbb{P}(\text{mask}(\tilde{\mathbf{h}}_\tau)), \quad (25)$$

$$\underline{\mathbf{x}} \leftarrow [\underline{\mathbf{x}}, (\mathbf{h}_{\text{embed}})_i]. \quad (26)$$

Here, i represents the index of vector $\tilde{\mathbf{h}}_\tau$. The masking rule operates as follows: during the very first sub-step within step t , all ports are masked to allow for vessel selection. From the second sub-step onward, the ports are unmasked while the vessels are masked. Ports that have already been visited in step t remain masked, except for the first port, as revisiting it indicates the completion of a service generation. Noted that the ‘‘begin of service’’ (BOS) embedding is only unmasked at the first sub-step ($\tau = 1$) of the initial step ($t = 1$).

Importantly, both \mathbf{x}_t and $\underline{\mathbf{x}}$ represent the same set of embeddings, capturing the vessels and ports selected at various stages. When carried over across different steps, it is referred to as \mathbf{x}_t , whereas when carried over across sub-steps within the same step, it is denoted as $\underline{\mathbf{x}}$.

The sub-steps in each action are generated autoregressively, with our neural network architecture applying the chain rule to factorize the probability of generating a service at step t as:

$$\pi_\theta(A_t|S_t) = \prod_{\tau=1}^{n_\tau} \mathbb{P}(A_t(\tau)|A_t(\tau' < \tau), S_t), \quad (27)$$

where n_τ represents the total number of sub-steps within step t . The term $A_t(\tau)$ refers to the selection made at sub-step τ , while $A_t(\tau' < \tau)$ denotes all selections made in the previous sub-steps leading up to τ .

5 Policy Optimization

To optimize our policy network π_θ , we employ policy gradient methods, which iteratively refine the policy to maximize the reward. Policy gradient methods form a broad class of reinforcement learning algorithms that directly improve the policy π_θ by rewarding actions that lead to higher-value outcomes based on sampled trajectories. In this work, we utilize an enhanced variant of the standard policy gradient algorithm known as Proximal Policy Optimization (PPO) [Schulman et al. \[2017\]](#). PPO introduces a clipped surrogate objective that significantly enhances the stability of the learning process. Detailed hyperparameter settings for PPO are provided in [Appendix D](#).

6 Experiments

We conduct experiments to evaluate the performance of the proposed RL approach on the LINERLIB benchmark, demonstrating that the RL-based solution for the NDP is a promising alternative to MIP and heuristic methods. Notably,

Google ORTools¹ recently published their own benchmark results on the LINERLIB dataset, which we include in our comparison where applicable.

To focus on benchmarking the solution quality for the NDP, we use the MCF algorithm introduced in Appendix B to evaluate all network solutions, including those generated by our RL-based method, LINERLIB, and ORTools. For a consistent comparison, we decompose all multi-loop services, such as butterfly services, into simple services. Additionally, we relax the hard limit on the number of vessels, treating it as a soft constraint with an associated penalty, to further facilitate the comparison.

Unless otherwise stated, all experiments, including both training and inference, are performed on an A100 GPU. As described in Section 5, we employ the standard PPO algorithm for policy optimization during training. For detailed hyper-parameter settings, please refer to Appendix D.

6.1 Result on Baltic Instance

In this section, we evaluate the performance of our RL-based NDP approach on the Baltic instance from the LINERLIB dataset, which consists of 12 ports (i.e., vertices in the graph). Training and validation are conducted on separate datasets with a total of 16,000 instances, where the demand quantities were perturbed from the original LINERLIB Baltic instance (with a factor of $\pm 10\%$), while the origins and destinations of the demand remained unchanged. For a detailed description of the perturbation process, please refer to Appendix E. The test set consists of a single data point—the actual LINERLIB Baltic instance—to ensure a fair comparison with publicly available benchmark solutions. It is worth noting that we primarily report results from the encoder-decoder RL-based solution, as the encoder-only variant produced nearly identical results in this case.

Table 1 provides a detailed profit breakdown for the RL-based solution and compares it with the LINERLIB benchmark. Notably, if a solution utilizes fewer vessels than the available fleet, a profit is gained based on the time charter rate. Conversely, if the solution requires more vessels than available, additional costs are incurred.

Table 1: Profit (\$) breakdown of the RL-based NDP solution and LINERLIB solution.

	RL-based Solution	LINERLIB Solution
Revenue	3,688,028	3,687,260
Unused vessel profit	-12,596	6823
Vessel used	6.34	5.72
Vessel service cost	267,898	245,176
Voyage cost and fee	634,729	689,083
Handling and transshipment cost	2,116,377	2,109,876
Rejected demand penalty	380,000	389,000
Total net profit	276,428	260,948

The RL-based approach utilizes 2.03 Feeder 800 vessels (with a capacity of 800 FFEs) and 4.31 Feeder 450 vessels (with a capacity of 450 FFEs), while the LINERLIB solution employs 2.14 Feeder 800 vessels and 3.58 Feeder 450 vessels. Any over- or under-utilization of vessels is accounted for in the “unused vessel profit,” where the LINERLIB solution shows a positive profit, while the RL-based solution incurs a loss. However, the RL-based solution is able to satisfy more demand, resulting in a smaller penalty for rejected demand due to the higher vessel utilization. Despite these nuances, the RL-based solution ultimately achieves a higher net profit compared to the LINERLIB solution. For a visual comparison of the network designs, refer to Figures 8 and 9 in Appendix F, which show the networks produced by the RL-based and LINERLIB solutions, respectively.

6.2 Experiments on Other Instances

In this section, we explore the potential of using the RL-based NDP solution as an optimizer, where the training process of the RL agent functions as a traditional optimization solver. In this scenario, there is no distinction between the training and inference phases.

We extend the experiments to include two additional instances from LINERLIB: West Africa (WAF) and World Small. These instances contain 20 and 47 ports, respectively. A performance comparison between the two RL-based NDP approaches—encoder-only and encoder-decoder—and the benchmark solutions is presented in Table 2. Both

¹<https://developers.google.com/optimization/service/shipping/benchmarks/lsndsp>

LINERLIB and ORTools solutions are listed as benchmarks, with ORTools results being available only for the World Small instance. Please refer to Appendix G for a visual comparison of the networks designed by the RL-based solution and the LINERLIB solution.

To ensure a fair comparison, both the LINERLIB and ORTools solutions are evaluated using our MCF algorithm introduced in Appendix B.

Table 2: Max profit (in million \$) of RL-based NDP approaches on LINERLIB problem instances.

	Baltic (n=12)	WAF (n=20)	World Small (n=47)
Encoder-only (RL)	0.28	5.60	42.73
Encoder-decoder (RL)	0.29	5.60	42.32
LINERLIB	0.26	5.20	32.28
ORTools	N/A	N/A	40.10

Both RL-based NDP approaches yield higher profits compared to the benchmark solutions, demonstrating the clear potential of using the RL-based solutions as a viable alternative solver for the Network Design Problem.

6.3 Solve Time Comparison

Table 3 reports both the inference time and training time for the RL-based NDP solution on the LINERLIB instances we experimented on. The solve time is benchmarked with the LINERLIB solution on corresponding instances. Note that for the RL-based NDP solution, the inferences are all run on an Apple M2 CPU with 12 cores, while the trainings are conducted on an A100 GPU. We only report the encoder-only approach as the RL-based NDP solution in Table 3 given that the encoder-decoder approach yields an inference time similar to that of the encoder-only approach.

Table 3: Inference time (in seconds) of the proposed encoder-only approach on different LINERLIB instances.

	Baltic (n=12)	WAF (n=20)	World Small (n=47)
RL-based NDP inference	0.03	0.11	0.60
Environment simulation + RL-based inference	0.22	0.40	15.04
LINERLIB benchmark	300	900	10,800
RL training time (excluding inferences)	720	4000	360,000

It is important to clarify that the “RL-based NDP inference” time in Table 3 includes both the generation of the full set of services and the execution of the underlying MCF algorithm, whereas the “environment simulation + RL-based inference” time also accounts for the setup and updates of the environment. When comparing the inference time of the RL-based solution to the solve time of the LINERLIB solution, which uses a MIP solver, we observe an approximate 1000x speedup across the three problem instances tested.

6.4 Solution Robustness against Variations in the Problem Instance

In real-world applications beyond the academic scope of the LSNDP, schedulers often face disturbances on short notice that can have long-term impacts. Examples include trade wars, which affect demand quantities, or pirate activities, which influence the availability of certain ports or routes in network design. Consequently, having an algorithmic tool that can quickly generate optimal network designs for a variety of perturbed problem instances is of immense value.

In this part of the experiment, we evaluate the effectiveness of our RL-based NDP solution in handling a large set of problem instances perturbed from a common baseline. Additionally, we explore how enhancing the RL agent by exposing it to these perturbed instances during training improves its performance compared to the baseline RL agent, which is trained on a single problem instance (as used in Section 6.2). For the enhanced RL agent, training and validation are conducted on separate datasets totaling 80,000 instances, where demand quantities are perturbed by $\pm 10\%$ from the original LINERLIB Baltic instance (matching the perturbation level in Section 6.1). During inference, 100 different test instances are randomly generated, and for each instance, the RL agent produces 100 network designs. From these designs, the maximum profit (i.e., reward) is selected for each instance. The mean and standard deviation of these maximum profits across the 100 test instances are then reported.

To further examine the agent’s robustness, we increase the perturbation level in both the training and test datasets from 10% to 50%, assessing how much additional improvement the enhanced RL agent offers over the baseline. Notably, the

perturbation levels are kept consistent between training and testing; for instance, when evaluating performance on a dataset with 50% perturbation, the RL agent is trained on data with the same 50% perturbation.

The first row of Table 4 compares the mean profit value across 100 test instances with a 10% perturbation between the enhanced and baseline RL agents. On average, the enhanced RL agent trained on perturbed instances achieves a \$257,945 higher profit than the baseline RL agent trained on a single instance. The second row shows that with the demand quantity perturbation increased to 50%, both agents perform worse, but the profit uplift from the enhanced RL agent rises to \$401,005. It is important to note that these results are based on the encoder-decoder RL approach, though we expect similar trends for the encoder-only approach.

Table 4: Mean and standard deviation (in parenthesis) of profits in \$ over 100 test instances for enhanced RL agent and baseline RL agent at corresponding perturbation levels.

	Enhanced agent trained on perturbed instances		Baseline agent trained on single instance	
Test dataset (10% perturbation)	274,387.17	(5,835.62)	16,441.79	(25,430.68)
Test dataset (50% perturbation)	78,215.53	(34,042.51)	-322,789.09	(56,312.07)

6.5 Discussion

The experiments conducted in this section demonstrate the effectiveness of our RL-based NDP solution in two significant ways. Firstly, when evaluated on the Baltic instance from the LINERLIB dataset, the RL-based solution generates near-optimal results and compares favorably against the benchmark solutions. Secondly, when applied as an optimizer on previously unseen instances, such as the Baltic, West Africa (WAF), and World Small datasets, the RL-based approach continues to deliver competitive performance without the need for retraining. This motivates the use of reinforcement learning based methods to learn general, competitive policies that can potentially deliver high-quality solutions on new instances.

However, a few limitations should be noted, stemming from both the computational resources and the experimental setup. The most significant limitation arises from the underlying heuristic multi-commodity flow (MCF) algorithm, which serves as a key part of the reward function evaluator for the RL agent. Unfortunately, we do not have access to the state-of-the-art MCF implementations used by the benchmarks. This discrepancy between our MCF implementation and those used in the benchmarks means the associated NDPs are effectively different problems. As a result, we evaluate all benchmark solutions using our MCF implementation.

Additionally, it’s important to note that the NDP definition has been relaxed to better align with our RL-based approach. For instance, we consider only simple services in the network design and relaxed the hard limit on the number of vessels to a soft constraint with penalties. These modifications explain why the LINERLIB and ORTools solutions reported here may differ from those found in other literature. According to industry experts, these relaxations have only a limited impact on the solution quality. Nonetheless, tightening these constraints and preparing the solution for an end-to-end benchmark on the full LSNDP would be a valuable next step.

Our current experiments cover three out of the seven instances in the LINERLIB dataset. Expanding the experiments to include all instances, particularly the World Large instance (the largest in the dataset), would further test the scalability of the approach. Moreover, the perturbations in this study are limited to demand quantities. Extending the perturbations to include the origin and destination of the demand, the number of available vessels in each class, and the inclusion of specific ports would provide a more comprehensive demonstration of the solution’s generalizability.

7 Conclusion and Future Work

In this paper, we propose a model-free RL-based framework to address the network design aspect of the Liner Shipping Network Design Problem (LSNDP). By leveraging a heuristic-based multi-commodity flow (MCF) solver as part of the evaluator function, our approach can solve the LSNDP in an end-to-end fashion. This work marks the first attempt to approach LSNDP through a method distinct from traditional operations research (OR) techniques. Our framework demonstrates scalability with problem size and achieves competitive results on the LINERLIB benchmark. We have shown that our approach offers value in two key ways: it can rapidly generate near-optimal solutions for problem instances perturbed from the training data or be utilized as an optimizer, delivering effective performance on unseen problem instances without requiring prior training.

Our approach introduces a novel paradigm for solving LSNDP compared to conventional OR methods, which typically require equal computational effort for each new problem instance. In contrast, our method front-loads the computational work during the training phase, while enabling rapid inference to new instances. This makes the solution ideal for problems like LSNDP, where long-term plans are frequently disrupted by unexpected events or frequent data updates. In terms of real world applications, this enables rapid “tactical” changes by reacting to real world dynamics.

Looking ahead, there are several opportunities to enhance the encoder-decoder architecture. Replacing the LSTM with a transformer-based architecture could allow the network to handle larger and more complex use cases. Additionally, as mentioned in Section 6.3, the MCF algorithm accounts for a significant portion of the runtime during both training and inference. A faster MCF implementation would further improve training efficiency and reduce the training wall clock time. Exploring a Graph Neural Network (GNN)-based surrogate for MCF is another promising avenue to speed up reward function evaluation.

In terms of training strategies, there are several paths to explore. One interesting direction is the application of reward shaping, as discussed by Ng et al. [1999], to enhance training efficiency. Reward shaping provides additional signals to guide the RL agent toward an optimal policy, especially when only terminal rewards are available during policy exploration. Introducing penalties in the reward function could also help guide the agent’s behavior; for instance, adding a service length penalty could encourage the agent to select shorter routes that optimize transshipment usage.

Another area worth exploring is the inherent symmetry of the LSNDP, where the reward remains unchanged if ports are rotated within a service. Inspired by OR techniques, which often limit symmetry in the search space to improve solving speed, symmetry can also be leveraged in neural combinatorial optimization, as demonstrated by Kwon et al. [2020] and Kim et al. [2022]. Adapting these techniques to our RL framework could improve sample efficiency for LSNDP.

Other reinforcement learning algorithms that favor exploration and improve sample efficiency could potentially improve performance over PPO. For example, Soft Actor-Critic (SAC, Haarnoja et al. [2018]), an off-policy actor-critic algorithm, maximizes both expected reward and entropy, promoting more effective exploration. Additionally, Monte Carlo Tree Search (MCTS, Kocsis and Szepesvári [2006]) based methods when combined with neural networks Silver et al. [2017] have proved to be effective model-based approaches and have yielded superhuman performance in deterministic environments. RL algorithms designed to scale with problem size (Drori et al. [2020]) and generalize across a variety of instances (Fu et al. [2021]) may also align with broader business needs beyond LSNDP. Adapting these techniques to train models on smaller instances and transfer the learned policy to larger problems would be a valuable extension of this work.

Acknowledgement

We want to extend our heartfelt gratitude to Carl Mikkelsen, Jimmy Paillet, and Torkil Kollsker for their guidance in designing the multi-commodity flow heuristics used in this paper. Our deep appreciation also goes to Josh Zhang for his support in enhancing the algorithm’s efficiency. We are especially grateful to Sina Pakazad for generously providing the necessary computing resources. We acknowledge the insightful contributions from Henrik Ohlsson, Nikhil Krishnan, Sravan Jayanthi, and Fabian Rigterink during the early stages of problem formulation. Additionally, we are thankful for the project management support from Marius Golombeck, Abhay Soorya, and Mehdi Maasoumy. We would like to express our gratitude to C3.ai for their funding and platform infrastructure. If you are interested in providing funding or collaborating on future research in this area, we encourage you to reach out to C3.ai.

References

- A. Balakrishnan and C. V. Karsten. Container shipping service selection and cargo routing with transshipment limits. *European Journal of Operational Research*, 263(2):652–663, 2017.
- I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. In *International Conference on Learning Representations*, 2016.
- J. Bi, Y. Ma, J. Wang, Z. Cao, J. Chen, Y. Sun, and Y. M. Chee. Learning generalizable models for vehicle routing problems via knowledge distillation. *Advances in Neural Information Processing Systems*, 35:31226–31238, 2022.
- B. D. Brouer, J. F. Alvarez, C. E. Plum, D. Pisinger, and M. M. Sigurd. A base integer programming model and benchmark suite for liner-shipping network design. *Transportation Science*, 48(2):281–312, 2014.
- M. Christiansen, E. Hellsten, D. Pisinger, D. Sacramento, and C. Vilhelmsen. Liner shipping network design. *European Journal of Operational Research*, 286(1):1–20, 2020.
- I. Drori, A. Kharkar, W. R. Sickinger, B. Kates, Q. Ma, S. Ge, E. Dolev, B. Dietrich, D. P. Williamson, and M. Udell. Learning to solve combinatorial optimization problems on real-world graphs in linear time. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 19–24. IEEE, 2020.
- G. Fellek, A. Farid, G. Gebreyesus, S. Fujimura, and O. Yoshie. Graph transformer with reinforcement learning for vehicle routing problem. *IEEJ Transactions on Electrical and Electronic Engineering*, 18(5):701–713, 2023.
- Z.-H. Fu, K.-B. Qiu, and H. Zha. Generalize a small pre-trained model to arbitrarily large tsp instances. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 7474–7482, 2021.
- T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.
- H. Hu, X. Zhang, X. Yan, L. Wang, and Y. Xu. Solving a new 3d bin packing problem with deep reinforcement learning method. *arXiv preprint arXiv:1708.05930*, 2017.
- C. K. Joshi, T. Laurent, and X. Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019.
- E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.
- M. Kim, J. Park, and J. Park. Sym-nco: Leveraging symmetry for neural combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 35, pages 1936–1949, 2022.
- S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, 2019. URL <https://doc.rust-lang.org/book/>.
- L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.
- W. Kool, H. van Hoof, and M. Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2018.
- A. Krosgaard, D. Pisinger, and J. Thorsen. A flow-first route-next heuristic for liner shipping network design. *Networks*, 72(3):358–381, 2018.
- Y.-D. Kwon, J. Choo, B. Kim, I. Yoon, Y. Gwon, and S. Min. Pomo: Policy optimization with multiple optima for reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 33, pages 21188–21198, 2020.
- Q. Meng and S. Wang. Liner shipping service network design with empty container repositioning. *Transportation Research Part E: Logistics and Transportation Review*, 47(5):695–708, 2011.
- M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác. Reinforcement learning for solving the vehicle routing problem. *Advances in neural information processing systems*, 31, 2018.
- A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *International Conference on Machine Learning*, volume 99, pages 278–287, 1999.
- C. E. Plum, D. Pisinger, and M. M. Sigurd. A service flow model for the liner shipping network design problem. *European Journal of Operational Research*, 235(2):378–386, 2014.
- J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- D. Shintyakov. TSP-solver2: Greedy, suboptimal solver for the Travelling Salesman Problem. <https://pypi.org/project/tsp-solver2/>, 2017.

- D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017. URL <https://arxiv.org/abs/1712.01815>.
- K. Thun, H. Andersson, and M. Christiansen. Analyzing complex service structures in liner shipping network design. *Flexible Services and Manufacturing Journal*, 29:535–552, 2017.
- O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.
- S. Wang and Q. Meng. Liner shipping network design with deadlines. *Computers & Operations Research*, 41:140–149, 2014.

A LSNDP details

A.1 Dataset

Brouer et al. [2014] offers a comprehensive introduction to the LSNDP benchmark suite, LINERLIB. Here, we provide a brief overview to establish the context for the problem we aim to address using reinforcement learning. At a high level, a liner shipping network comprises a fleet of vessels V deployed across rotations or services S to satisfy a set of commodity demands D , normalized to a weekly frequency. Visualizing the network as a graph, ports can be seen as vertices, with edges E representing the connections between them. Each service $s \in S$ involves a rotation through a sequence of ports s_P , is assigned a specific subset of vessels s_V , and includes a set of legs s_E that define the route. Below, we provide a more detailed breakdown of each of these elements.

LINERLIB includes a predefined set of ports P that vessels can access. Each port p within this set is characterized by the following features:

p	Port ID, represented by UNLOCODE.
p_f	Fixed cost per port call, the cost in USD for each vessel call at this port.
p_v	Variable cost per port call, the additional cost in USD per FFE for visiting this port, based on the vessel's capacity.
p_t	Transshipment cost per FFE, the cost in USD per FFE for transferring cargo across different services at this port.

A fleet of vessels V contains different vessel classes v^F , each with different capacities and characteristics. Each vessel class has a finite number of vessels available for deployment. A vessel $v \in v^F$ is characterized by the following features:

v_{cap}	Capacity, the maximum number of FFEs the vessel can carry at once.
v_n	Quantity, the total number of available vessels of class v .
v_{TC}	TC rate, the daily cost of renting or operating the vessel.
v_s	Design speed, the vessel's standard sailing speed.
v_{fs}	Fuel consumption at design speed, the vessel's daily fuel consumption (converted to \$) when sailing at design speed.
v_{fi}	Fuel consumption while idling, the vessel's daily fuel consumption (converted to \$) when idle at the port.
v_{Suez}	Suez fee, the fee for passing through the Suez Canal.
v_{Panama}	Panama fee, the fee for passing through the Panama Canal.

Each port in p also includes data on latitude and longitude coordinates (which we omitted earlier for brevity). The LINERLIB dataset provides distance information, including whether the route passes through the Panama or Suez canals. This distance data corresponds to the edge $e \in E$ in the graph and is described by the following features:

e_o	Origin port, the Port ID in UNLOCODE.
e_d	Destination port, the Port ID in UNLOCODE.
e_{dist}	Distance, the distance between the origin port and the destination port, measured in nautical miles.
e_{Suez}	Suez traversal, a flag indicating whether the sailing route passes through the Suez Canal. A value of 1 signifies the route uses the Suez Canal; 0 otherwise.
e_{Panama}	Panama traversal, a flag indicating whether the sailing route passes through the Panama Canal. A value of 1 signifies the route uses the Panama Canal; 0 otherwise.

At last, each commodity demand $d \in D$ is characterized by the following features:

d_o	Origin port, the Port ID in UNLOCODE.
d_d	Destination port, the Port ID in UNLOCODE.
d_R	Revenue, generated per unit FFE transported.
d_q	Quantity, demand quantity in FFE per week.
Y_d	Penalty if rejected, penalty for rejection of this demand, which is set to \$1000.

Note that we have only listed the dataset elements relevant for solving the LSNDP with transshipment, rejected demand, and fractional vessel assignments. For a complete description of the dataset, please refer to Brouer et al. [2014]. Throughout this paper, when we refer to an "instance", we mean a specific LSNDP setup with a defined set of commodities, available ports, edges, and fleet, which collectively determine the characteristics of the shipping network.

A.2 Multi Commodity Flow

The maximum profit Multi-Commodity Flow Problem seeks to determine the optimal flow of multiple commodities through a capacitated network to maximize total profit. Each commodity has a specific origin and destination and moves through the network's edges, constrained by capacity limits. Additionally, each unit of flow for a commodity may generate a defined revenue. The goal is to allocate flows for all commodities in a way that maximizes the overall profit while adhering to the network's capacity constraints. The capacity limits of the network are defined by the designed rotations or services from the associated network design problem, which will be discussed later. The capacity constraints on each edge of the network are derived from the capacities of the vessels assigned to those routes.

Here, we omit the specific details of the constraints and focus only on the objective function. For a comprehensive description of the complete problem formulation, please refer to [Brouer et al. \[2014\]](#).

$$\text{Maximize } \eta = R_{\text{total}} - C_{\text{reject}} - C_{\text{handle}} - C_{\text{NDP}}, \quad (28)$$

where η is the profit, R_{total} represents the total revenue generated, C_{reject} represents the penalty associated with rejected demand, and C_{handle} denotes the handling costs. Detailed descriptions of these terms are provided in the equations below. Note that C_{NDP} is the fixed cost of establishing all services in the network, independent of the decisions made within the multi-commodity flow problem. This fixed cost is determined by the network design and will be discussed in detail later.

$$R_{\text{total}} = \sum_{d \in D} d_R \left(\sum_{\forall e | e_d = d_d} f_e^d \right), \quad (29)$$

$$C_{\text{reject}} = Y_d \sum_{d \in D} \left(d_q - \sum_{\forall e | e_d = d_d} f_e^d \right), \quad (30)$$

$$C_{\text{handle}} = \sum_{p \in P} p_l \left(\sum_{\forall e | e_d = d_d = p} f_e^d + \sum_{\forall e | e_o = d_o = p} f_e^d \right) + \sum_{p \in P} p_t \sum_{\substack{\forall e', e'' \in S_E \\ e'_d = p, e''_o = p, e'_d \neq d_d}} (f_{e'}^d - f_{e''}^d). \quad (31)$$

Here, f_e^d is a decision variable within MCF, which represents the quantity of commodity demand d that flows through edge e , while e' and e'' refer to two edges within the same service. The remaining notation is detailed in [Appendix A.1](#). It is important to note that the handling cost, as described in [Eq. 31](#), has two components: the first is the cost associated with onloading and offloading, and the second is the transshipment cost. It is important to note that, in contrast to the fixed cost C_{NDP} , the terms in [Eqs. 29, 30, and 31](#) represent variable costs, with revenue broadly considered as a form of negative cost.

The MCF problem is known to be NP-hard (see [Theorem 6.2 in Brouer et al. \[2014\]](#)). Rather than solving it using a MIP formulation, we employ a fast, greedy heuristic-based approach, which is detailed in [Appendix B](#).

A.3 Network Design Problem

The network design problem (NDP) focuses on identifying the optimal set of services for a given LSNDP instance to maximize the overall profitability of the shipping network. However, the ultimate profitability of the network is determined by the Multi-Commodity Flow (MCF) solution, as discussed previously. The primary objective of the NDP is to develop a network design that defines the capacities on the edges of the services, which are used as constraints in MCF. The fixed cost associated with the designed network corresponds to the C_{NDP} term in [Eq. 28](#) and is calculated as follows:

$$C_{\text{NDP}} = C_{\text{service}} + C_{\text{unused}} + C_{\text{voyage}}, \quad (32)$$

where C_{service} represents the vessel service cost, accounting for the total cost of renting or operating all vessels assigned to the services. C_{unused} captures the cost (or profit) of unused vessels. If the generated services do not utilize all available vessels, the remaining vessels can be rented out at the time charter (or TC) rates. Conversely, if the services require more vessels than are available, additional vessels must be acquired at the same rate. The term C_{voyage} refers to the voyage cost, which includes fuel costs, port calling costs, and canal fees associated with operating the vessels to support

the services. The detailed breakdown of these terms is provided in the equations below:

$$C_{\text{service}} = \sum_{s \in S} \sum_{v \in s_V} n_{v,s} \cdot v_{\text{TC}}, \quad (33)$$

$$C_{\text{unused}} = - \sum_{v \in V} \left(v_n - \sum_{r \in R} n_{v,r} \cdot v_{\text{TC}} \right), \quad (34)$$

$$\begin{aligned} C_{\text{voyage}} = & \sum_{s \in S} \sum_{p \in s_P} \sum_{v \in s_V} (p_f + p_v \cdot v_{\text{cap}}) \cdot n_{v,s} + \sum_{s \in S} \sum_{v \in s_V} \left(\frac{\sum_{e \in s_E} e_{\text{dist}}}{v_s} \cdot v_{\text{fs}} + \sum_{p \in s_P} 1 \cdot v_{\text{fi}} \right) \cdot n_{v,s} \\ & + \sum_{s \in S} \sum_{v \in s_V} \sum_{e \in s_E} (e_{\text{suez}} \cdot v_{\text{suez}} + e_{\text{panama}} \cdot v_{\text{panama}}). \end{aligned} \quad (35)$$

Here, C_{voyage} accounts for various voyage-related expenses, including fixed and variable port fees, fuel consumption during sailing, and costs associated with passing through the Suez and Panama canals. The variable $n_{v,s}$ represents the number of vessels in a given class that are assigned to service s , making it a key decision variable in the NDP. The remaining notation is detailed in Appendix A.1. Since the problem setup assumes that vessels operate strictly at their designed speed and allows for fractional vessel assignments, the required number of vessels can be calculated as a function of the total distance of the service and the vessel's designed speed. As a result, there is no expected idle time for the vessels except for the required 1 day at each port.

To integrate the NDP and MCF, we first generate a network schedule through the NDP, which is subsequently used as input for the MCF to calculate the associated revenues and penalties. The combined objective value thus includes both the static network costs derived from the NDP and the variable flow costs calculated from the MCF. This combined objective serves as the reward function, which is used to evaluate the performance of our complete algorithm.

B Heuristic MCF Details

As outlined in Appendix A.2, the Multi-Commodity Flow (MCF) problem seeks to identify optimal cargo routes that maximize profit within a capacitated network. Instead of solving the MCF to optimality using a Mixed-Integer Programming (MIP) approach, we propose a faster, heuristic-based method that employs a greedy sequential commodity flow strategy, building on a graph representation of the liner shipping network.

However, directly using the original graph representation described in Section 4 presents challenges for the heuristic MCF. Once we get into the MCF phase with the original representation, the graph only includes edges between ports that are already connected by established services from the Network Design Problem (NDP) phase. As shown on the left side of Fig. 4, the edge weights w represent the variable cost of moving an additional FFE of cargo, while edge capacities define the maximum number of FFEs that can flow through those edges. Importantly, in this original representation, $w = 0$, since it only accounts for the variable cost of transporting cargo between ports, and it does not capture the internal dynamics of variable costs within a port, such as handling costs.

To effectively implement the heuristic MCF algorithm, the graph representation must be expanded to include these intra-port dynamics. Specifically, the handling costs — such as onloading, offloading, and transshipment — need to be incorporated as edge weights on the graph, allowing for an accurate representation of the network's cost structure.

Among the variable cost terms, only the handling cost C_{handle} (as defined in Eq. 31) is considered in this expanded graph representation. The revenue and rejected demand penalty are excluded for the following reasons: revenue is tracked separately within the heuristic MCF algorithm, which will be discussed in detail later, and the rejected demand penalty is uniform across all commodities, making it irrelevant to the heuristic MCF algorithm where the focus is on balancing trade-offs between commodities. Additionally, the fixed costs, as outlined in Eqs. 33, 34, and 35, are not included at this stage. These costs are already determined during the Network Design Problem (NDP) phase, and decisions made during the MCF phase will not affect them.

The right side of Fig. 4 illustrates this expanded graph representation. A key component of this expansion is the introduction of proxy nodes to represent port-service pairs for each port. For example, if ports p and q are visited by rotations s_1, s_3 and s_7 , proxy ports p_1, p_3 and p_7 are created for p , and similarly q_1, q_3 and q_7 for q .

In the expanded graph, edges connecting p to its proxy nodes (p, p_s) , where $s \in \{1, 3, 7\}$, have weights representing the onloading and offloading costs at port p . These edges are assigned infinite capacity, reflecting that there are no logistical constraints on the volume of goods that can be handled at a port. Additionally, edges are introduced between different proxy nodes $p_{s'}$ and $p_{s''}$, where $s' \neq s''$, with infinite capacities. The weights on these edges correspond to transshipment costs at port p .

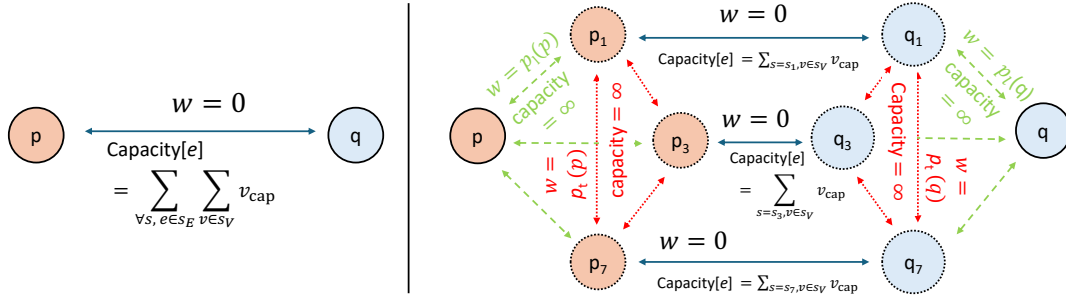


Figure 4: Expanded graph representation for an edge connecting ports p and q . The expanded representation on the right side of the figure with proxy nodes fully captures all dynamics of commodity shipping.

Finally, the original direct connection between ports p and q is now decomposed into edges (p_s, q_s) for each service s . The capacity of these edges is determined by the capacity of the respective service, while the weights remain zero, as there is no additional cost for shipping commodities between ports once they are loaded onto a service, as long as the service capacity is not exceeded.

Algorithm 1: Heuristic MCF with demand revenue prioritization

- 1: Input: Proposed services S , Graph $G(P, E)$, Demand \mathbf{D} , Revenue per demand \mathbf{R}
 - 2: Initialize: Commodity flows $f_e^d = []$, Missed (rejected) demand $\mathbf{D}_m = []$, Service capacity projected to edges: $\text{Capacity}[e], \forall e \in E$;
 - 3: Descending sort \mathbf{D} based on \mathbf{R} ;
 - 4: **for** $d = 1, \dots, D$ **do**
 - 5: Initialize the remaining demand $d_r = d$;
 - 6: Get all available paths \mathbf{T} from d_o to d_d ;
 - 7: Ascending sort \mathbf{T} based on **marginal unit cost** (from C_{handling});
 - 8: **for** $t = 1, \dots, \mathbf{T}$ **do**
 - 9: Get path capacity: $t[\text{capacity}] = \min(\{\text{Capacity}[e] \text{ for } e \text{ in } t.\text{edges}\})$
 - 10: Flow quantity: $q = \min\{d_r, t[\text{capacity}]\}$
 - 11: $d_r \leftarrow d_r - q$;
 - 12: Append $(d, e, q), \forall e \in t$ to f_e^d ;
 - 13: Update remaining edge capacities: $\text{Capacity}[e], \forall e \in t$;
 - 14: **if** $d_r = 0$, **break**; **end if**
 - 15: **end for**
 - 16: **if** $d_r > 0$, Append d_r to \mathbf{D}_m ; **end if**
 - 17: **end for**
 - 18: Return f_e^d, \mathbf{D}_m
-

With the expanded graph representation of the shipping network, we can now develop a greedy sequential commodity flow strategy. For each commodity demand d with origin d_o and destination d_d , we use Dijkstra's shortest path algorithm to find the least costly path t between d_o and d_d , where the edge weights represent variable costs instead of distances (as illustrated in Fig. 4). Once the cheapest path is identified, we ship a quantity equal to the path's capacity along this route. The path capacity is defined as the minimum capacity among all edges within the path. After shipping, the remaining capacities of all edges in path t are updated to account for the flow of d . This process is repeated for any remaining demand of d until no paths with non-zero capacities exist between d_o and d_d .

This method is applied sequentially across all commodity demands \mathbf{D} , with the order of processing determined by ranking the demands in descending order of their revenue per FFE. The pseudocode for this approach is outlined in Algorithm 1.

After the MCF generates the commodity flow patterns and the corresponding handling costs C_{handle} (defined in Eq. 31), we calculate the total revenue R_{total} and the penalties for missed (rejected) demand C_{reject} , using Eqs. 29 and 30, respectively.

It is worth mentioning that our implementation of the MCF is written in Rust (Klabnik and Nichols [2019]) to ensure high performance. Since the MCF algorithm is executed multiple times during each training iteration, a low-latency solution is essential for maintaining efficiency.

C MDP Details

Here, we represent the Liner Shipping Network Design Problem (LSNDP) as a Markov Decision Process (MDP). Figure 5 provides a high-level overview of this representation. The action A_t consists of selecting a vessel and

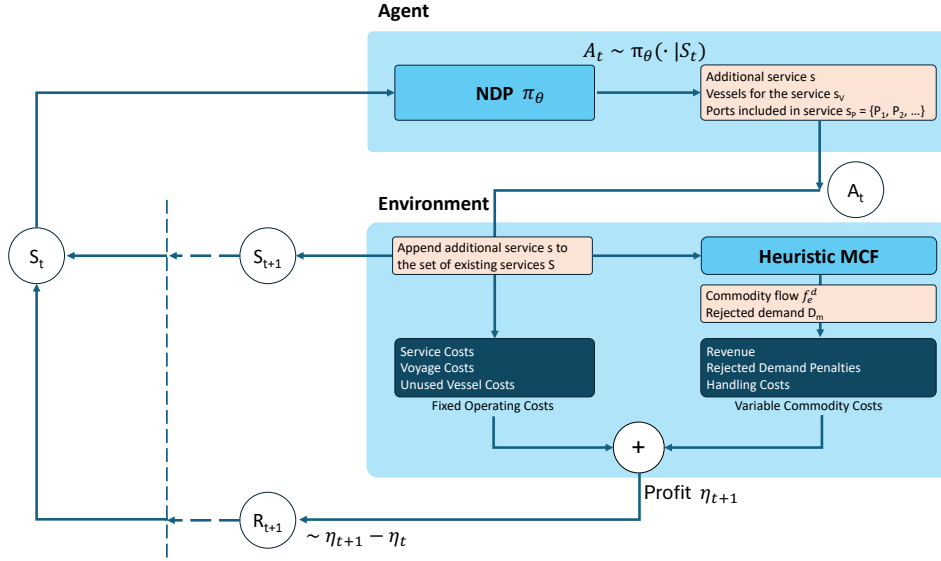


Figure 5: MDP representation of the LSNDP

determining the sequence of ports for the service s , as described in Eq. 3. The selected service s is then appended to the existing set of services S . This updated set of services is used to directly compute the fixed operating costs. Simultaneously, the updated services are fed into the heuristic MCF algorithm to calculate the variable costs, which include revenue (treated as a negative cost). These fixed and variable costs are summed to compute the total profit η_{t+1} .

The reward R_{t+1} , defined in Eq. 1, is then calculated, where both η_{t+1} and η_t are derived from Eq. 28 at their respective steps. To enhance the stability of the training process, the rewards at each step are scaled by the initial reward η_1 , as shown in Eq. 36 below:

$$R_{t+1} = \frac{\eta_{t+1} - \eta_t}{\eta_1}. \quad (36)$$

The full MDP process can be summarized in Algorithm 2 below.

Let's delve deeper into the state representation, denoted by S_t . The state at step t is composed of two main components: the vessel state $S_{v,t}$, and the graph state $S_{g,t}$, as defined in Eqs. 4 and 5, respectively. The graph state includes both port features \mathbf{f}_p and edge features \mathbf{f}_e .

Port Features: The port feature vector \mathbf{f}_p can be expressed as:

$$\mathbf{f}_p = [\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_{P+1}] \in \mathbb{R}^{P \times 2}, \quad (37)$$

where each element \mathbf{p}_p represents the total incoming and outgoing demand on port p :

$$\mathbf{p}_p = \left[\sum_{d_d=p} d_q, \sum_{d_o=p} d_q \right] \in \mathbb{R}^2. \quad (38)$$

Here, the first element of \mathbf{p}_p captures the total demand destined for port p , and the second element captures the total demand originating from port p . Additionally, P represents the total number of ports in the network. The last element, \mathbf{p}_{P+1} , represents a global node that remains empty, i.e., $\mathbf{p}_{P+1} = [0, 0]$.

Edge Features: The edge feature matrix \mathbf{f}_e is composed of both static and dynamic features:

$$\mathbf{f}_e = \begin{bmatrix} \mathbf{f}_e^s \\ \mathbf{f}_e^d \end{bmatrix} \in \mathbb{R}^{D_e \times E}, \quad (39)$$

where E is the number of edges, and $D_e = 6 + |S|$ is the total number of edge features.

The static features $\mathbf{f}_e^s \in \mathbb{R}^{4 \times E}$ consist of four attributes that do not change over time: the origin port index, the destination port index, the distance between the origin and destination, and the revenue generated per unit demand flowing through the edge.

The dynamic features $\mathbf{f}_e^d \in \mathbb{R}^{(2+|S|) \times E}$ include time-varying attributes: the remaining unsatisfied demand between the origin and destination, and the remaining vessel capacity on the edge. Additionally, $|S|$ number of binary indicators track whether an edge is included in service s , where a value of 1 indicates inclusion in the service and 0 otherwise.

Vessel Features: The vessel state \mathbf{v}_t represents the characteristics of each vessel class. The dimensionality of the vessel features $D_v = 11$, corresponding to the columns listed in Table 3 of [Brouer et al. \[2014\]](#), captures key vessel information at each step t .

Algorithm 2: Environment Step Function (from step t to $t + 1$)

Input: Action A_t , Previous state S_t , Remaining vessels V , Total demand D , Services S^* , Profit history $[\eta_0, \dots, \eta_t]$

Output: Next state S_{t+1} , Reward R_{t+1} , Done flag

Execute Action and Update State Value:

- Add the action to services, and reduce the remaining vessels:

$$\begin{aligned} (S^*, V) &\leftarrow S_t \\ S^* &\leftarrow S^* + A_t, \\ V &\leftarrow V - A_t, \\ S_{t+1} &\leftarrow (S^*, V) \end{aligned}$$

Run MCF Algorithm:

- Execute Algorithm 1 with current services S^* on \mathbf{D} :

$$f_e^d, \mathbf{D}_m, \text{Capacity}(E) \leftarrow \text{env.mcf}(S^*, \mathbf{D})$$

Compute Cost:

- Calculate the profit using the provided function:

$$\eta_{t+1} \leftarrow \text{env.get_profit}(S^*, f_e^d, \mathbf{D}_m, V)$$

- Append the profit to the profit history:

$$[\eta_1, \dots, \eta_t] \leftarrow \eta_{t+1}$$

Check for Termination:

- If all vessel counts are below 0 or remaining demands are 0, terminate:

$$\text{If } (v_n < 0 \forall v \in V) \text{ or } (\mathbf{D}_m = 0) \Rightarrow \text{terminate}$$

- Else, continue the episode

Calculate Reward:

- Compute the reward based on the change in profit:

$$R_{t+1} \leftarrow \frac{\eta_{t+1} - \eta_t}{\eta_1}$$

return $S_{t+1}, R_{t+1}, \text{Done}$;

D Problem Size and Hyper-parameter Setting

In this section, we summarize the problem size of each LINERLIB instance and the hyper-parameter settings of our approaches in Table 5.

Table 5: Hyper-parameter settings for encoder / decoder etworks and PPO.

Hyperparameter	Description	Range of Values
Neural Networks		
Hidden layer	Number of neurons in the hidden layer	512
Transformer head	Number of attention heads in the transformer	8
Transformer layer	Number of transformer layers	3
GAT layer	Number of Graph Attention Network layers	3
LSTM layer	Number of LSTM layers	1
PPO		
Learning rate	Constant learning rate for optimizer	[1e-4, 3e-4]
Environment	Number of parallel environments	[8, 16]
Step	Number of steps per environment per update	[50, 100]
Discount factor	Discount factor for future rewards	1
TD lambda	Lambda parameter for TD learning	0.9
Mini-batch size	Size of mini-batches for training	[64, 128]
Update epoch	Number of epochs per update	10
Clip coefficient	Clipping coefficient for PPO	[0.15, 0.25]
Target KL	Target Kullback-Leibler divergence	0.1
Entropy coefficient	Coefficient for entropy regularization	[0.01, 0.1]
Value function coef	Coefficient for value function loss	0.5

We use the **AdamW**² optimizer on Pytorch, with default values for β_1, β_2 and weight decay. Our environment is implemented with the **Gymnasium** Interface³.

²<https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html>

³<https://github.com/Farama-Foundation/Gymnasium>

E Perturbation Profiles

In this section, we discuss the perturbations applied to the demand quantities in the Baltic instance. Figures 6 and 7 illustrate the $\pm 10\%$ and $\pm 50\%$ perturbations, respectively. In both figures, the blue bars represent the original demand quantities from the Baltic instance, while the black error bars indicate the range of perturbation. The x-axis shows the origin and destination ports for each demand.

The 10% and 50% perturbations correspond to the standard deviation of the distribution used to generate new demand values. For example, when generating a perturbed instance with 10% perturbation, the new demand quantities are randomly sampled from a distribution where the original demand d_q is the mean, and the standard deviation is $10\% \cdot d_q$. Additionally, note that the samples are truncated at 0 to prevent the generation of instances with negative demand values.

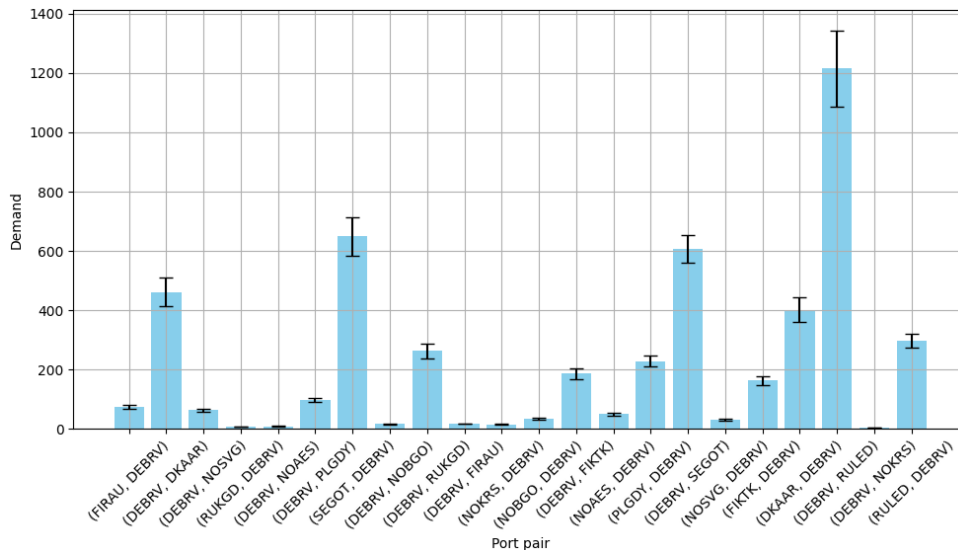


Figure 6: Perturbed demand quantities (in FFEs) for the Baltic instance with a perturbation level of 10%.

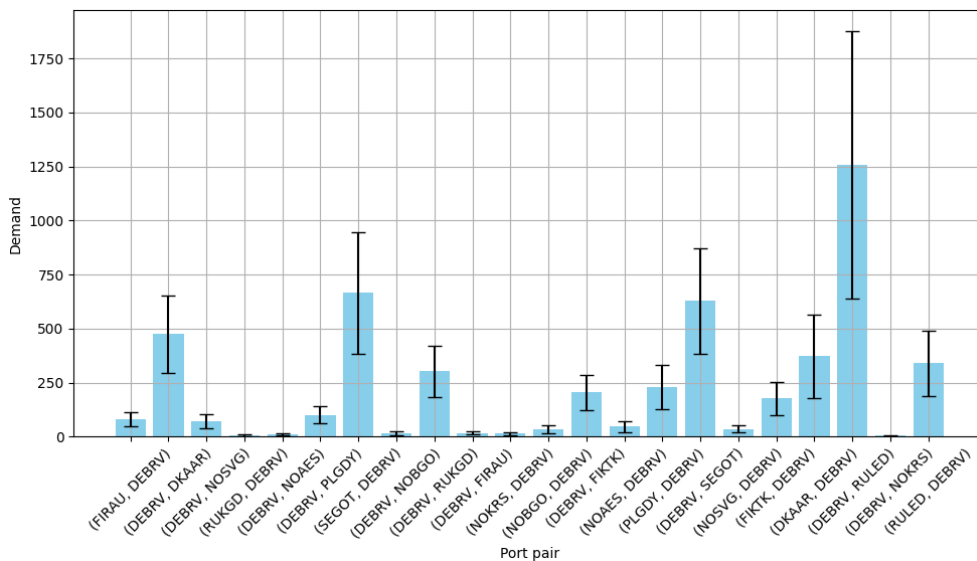


Figure 7: Perturbed demand quantities (in FFEs) for the Baltic instance with a perturbation level of 50%.

F Network Design Comparison for the Baltic Instance

Figure 8 illustrates the network design for the Baltic instance generated by the RL agent using the encoder-decoder approach. This design corresponds to the experimental setup described in Section 6.1, where training and validation were performed on a dataset comprising 16,000 instances, with demand quantities perturbed by $\pm 10\%$ from the original LINERLIB Baltic instance.

Figure 9 shows the network design for the Baltic instance produced by the LINERLIB solution. A comparison of the two figures reveals that the RL-based design covers the same 8 out of 12 ports as the LINERLIB solution. Additionally, both designs propose 5 simple services that follow a similar pattern.

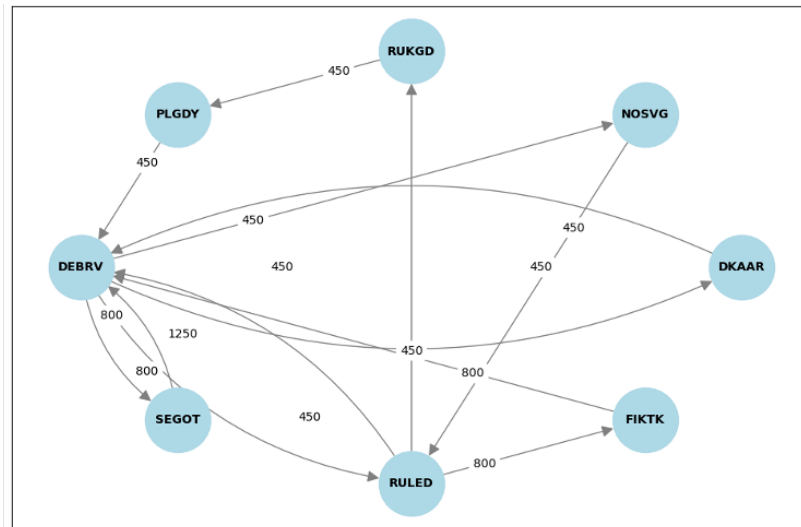


Figure 8: Network design for Baltic instance obtained by the RL-based approach. Ports are connected by vessels with specified capacity. The design uses 2.03 vessels of capacity 800 and 4.31 vessels of capacity 450. The total net profit is \$276,428.

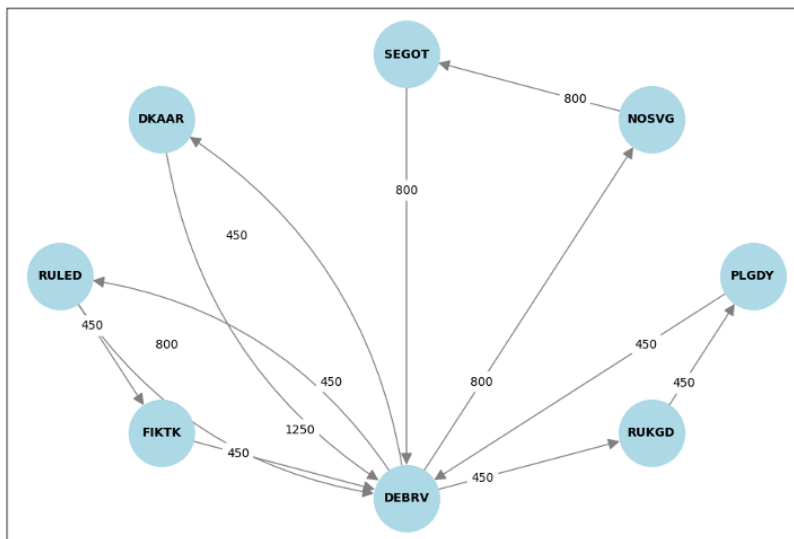


Figure 9: Network design for Baltic instance obtained by the LINERLIB solution. Ports are connected by vessels with specified capacity. The design uses 2.14 vessels of capacity 800 and 3.58 vessels of capacity 450. The total net profit is \$260,948.

G Network Design Comparison for the WAF and World Small instances

In this section, we utilize the RL-based NDP solution as an optimizer, where the RL agent’s training process functions similarly to a traditional optimization solver, eliminating the distinction between training and inference phases (as outlined in Section 6.2). We compare the network designs generated by our encoder-decoder approach and the LINERLIB solution for the WAF instance in Figure 10 and Figure 11, respectively. Additionally, the schedule outputs for the World Small instance from both our encoder-decoder approach and the LINERLIB solution are compared in Table 6 and Table 7. Finally, we compare vessel usage between the two approaches for the World Small instance in Table 8. Even though we obtain different network designs from the LINERLIB solution, the total net profits of our RL-based approach for both WAF and WorldSmall instances are higher than the LINERLIB benchmark, which shows the capability of our RL-based approach being used as an optimizer.

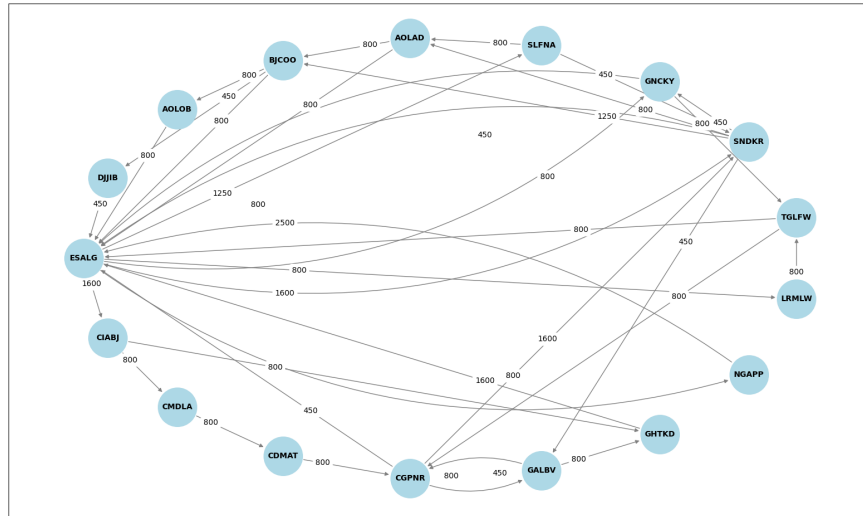


Figure 10: Network design for WAF instance obtained by running our encoder-decoder approach. Ports are connected by vessel with specified capacity. It uses 31.68 vessels of capacity 800 and 14.05 vessels of capacity 450. The total net profit is \$5,596,382.

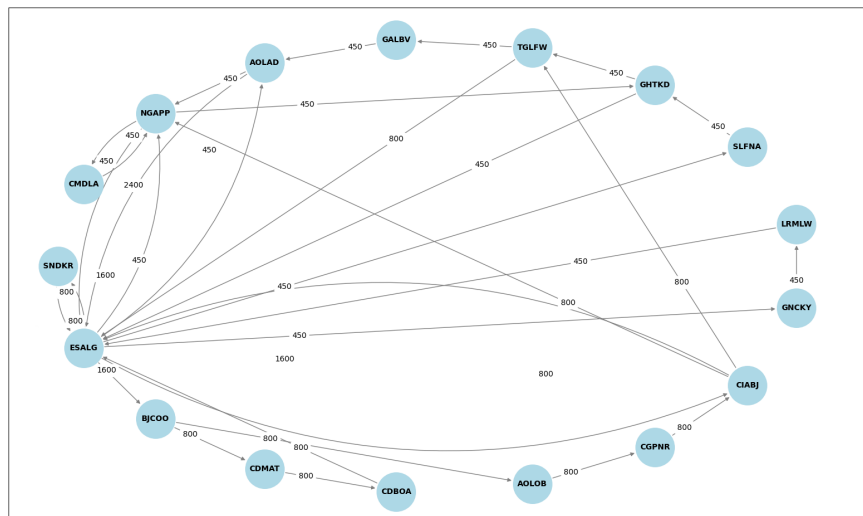


Figure 11: Network design for WAF instance obtained by the LINERLIB solution. Ports are connected by vessel with specified capacity. It uses 21.34 vessels of capacity 800 and 12.80 vessels of capacity 450. The total net profit is \$5,202,534.

Table 6: Schedule output for World Small instance obtained by our encoder-decoder approach.

Schedule	Vessel capacity	Number of vessels
Schedule 1 CNYTN → ITGIT → BEANR → PAMIT → USMIA → NGAPP → HKHKG → MAPTM → CNYTN	450	24.11
Schedule 2 CNYTN → TWKHH → NZAKL → CNSHA → MYTPP → AEJEA → INNSA → SAJED → ZADUR → KEMBA → HKHKG → LKCMB → OMSLL → MYPKG → CNYTN	800	19.29
Schedule 3 CNYTN → TWKHH → NZAKL → PABLB → CNSHA → MYTPP → LKCMB → PKBQM → HKHKG → MYPKG → KEMBA → INNSA → SGSIN → CNYTN	800	19.81
Schedule 4 CNYTN → TWKHH → NZAKL → PABLB → CLSAI → JPYOK → CNSHA → AEJEA → PKBQM → KEMBA → ZADUR → OMSLL → SGSIN → LKCMB → MYPKG → INNSA → AOLAD → MYTPP → HKHKG → SAJED → TRAMB → CNYTN	1200	28.98
Schedule 5 CNYTN → TWKHH → NZAKL → CLSAI → PABLB → ECGYE → PAMIT → USEWR → USMIA → BEZEE → BEANR → NLRTM → DEHAM → ITGIT → EGPSD → MAPTM → ESBCN → CAMTR → DEBRV → GBFXT → TRAMB → SAJED → INNSA → SGSIN → AEJEA → LKCMB → OMSLL → KEMBA → AOLAD → GHTKD → UYMVD → MYTPP → MYPKG → PKBQM → ZADUR → HKHKG → NGAPP → CNYTN	1200	41.42
Schedule 6 CNYTN → HKHKG → DEBRV → NLRTM → GBFXT → USCHS → ITGIT → ESALG → NGAPP → BRSSZ → ZADUR → AEJEA → MYTPP → CNSHA → CNYTN	2400	17.56
Schedule 7 CNYTN → HKHKG → CNTAO → CNSHA → ITGIT → EGPSD → MYTPP → CNYTN	2400	7.38
Schedule 8 CNYTN → HKHKG → JPYOK → CNSHA → MYTPP → ITGIT → EGPSD → SAJED → LKCMB → CNYTN	2400	8.12
Schedule 9 CNYTN → HKHKG → JPYOK → KRPUS → CNSHA → MYTPP → ITGIT → EGPSD → SAJED → AEJEA → INNSA → LKCMB → CNYTN	2400	9.18
Schedule 10 CNYTN → HKHKG → CNSHA → JPYOK → USLAX → CNTAO → KRPUS → TWKHH → CNYTN	2400	6.45
Schedule 11 CNYTN → HKHKG → CNTAO → KRPUS → USLAX → CAVAN → JPYOK → CNSHA → TWKHH → NGAPP → GHTKD → CNYTN	2400	14.10
Schedule 12 CNYTN → HKHKG → CNTAO → CNSHA → USLAX → CAVAN → JPYOK → KRPUS → TWKHH → AUBNE → CNYTN	2400	9.80
Schedule 13 CNYTN → HKHKG → TWKHH → CNSHA → CNTAO → AEJEA → INNSA → OMSLL → MYTPP → CNYTN	2400	6.60
Schedule 14 CNYTN → MYTPP → ESALG → GBFXT → DEBRV → NLRTM → NGAPP → ZADUR → HKHKG → TRAMB → BEANR → BEZEE → ITGIT → SAJED → LKCMB → CNYTN	4200	18.23
Schedule 15 CNYTN → GHTKD → USCHS → PAMIT → USMIA → NLRTM → AEJEA → MYTPP → ITGIT → GBFXT → DEBRV → MAPTM → CNYTN	4200	19.41
Schedule 16 CNYTN → ZADUR → USCHS → PAMIT → USMIA → USEWR → BRSSZ → DEBRV → ESALG → GHTKD → NGAPP → CNYTN	4200	17.84
Schedule 17 CNYTN → CNSHA → MYTPP → CNYTN	4200	2.07
Schedule 18 CNYTN → CNSHA → CNYTN	4200	0.88
Schedule 19 CNYTN → CNTAO → CNSHA → ZADUR → NGAPP → NLRTM → DEHAM → DEBRV → BEANR → GBFXT → EGPSD → MYTPP → CNYTN	7500	11.07

Table 7: Schedule output for World Small instance obtained by the LINERLIB solution.

	Schedule	Vessel capacity	Number of vessels
Schedule 1	NGAPP → GHTKD → NGAPP	450	0.60
Schedule 2	ZADUR → AOLAD → MAPTM → GHTKD → ZADUR	450	6.68
Schedule 3	OMSLL → SAJED → KEMBA → LKCMB → INNSA → OMSLL	450	4.73
Schedule 4	AEJEA → INNSA → AEJEA	450	1.46
Schedule 5	SAJED → KEMBA → SAJED	450	2.57
Schedule 6	INNSA → LKCMB → KEMBA → INNSA	450	3.31
Schedule 7	PALBL → ECGYE → PALBL	450	1.10
Schedule 8	PAMIT → USCHS → ECGYE → PAMIT	450	2.97
Schedule 9	LKCMB → MYTPP → TWKHH → CNTAO → HKHKG → MYPKG → INNSA → LKCMB	800	5.33
Schedule 10	PKBQM → SAJED → OMSLL → INNSA → PKBQM	800	2.73
Schedule 11	MAPTM → USEWR → USLAX → PALBL → PAMIT → BRSSZ → NGAPP → ESALG → MAPTM	800	12.15
Schedule 12	ITGIT → ESALG → ITGIT	800	1.18
Schedule 13	EGPSD → TRAMB → EGPSD	800	0.99
Schedule 14	ITGIT → TRAMB → ITGIT	800	1.15
Schedule 15	PKBQM → LKCMB → AEJEA → PKBQM	800	2.55
Schedule 16	INNSA → AEJEA → INNSA	800	1.30
Schedule 17	JPYOK → KRPUS → CNTAO → CAVAN → SGSIN → MYTPP → AEJEA → SAJED → OMSLL → MYPKG → HKHKG → CNYTN → TWKHH → CNSHA → JPYOK	1200	11.06
Schedule 18	EGPSD → SAJED → MYPKG → HKHKG → MYTPP → LKCMB → OMSLL → ESBCN → ITGIT → EGPSD	1200	6.75
Schedule 19	CNSHA → JPYOK → CAVAN → CNTAO → CNSHA	1200	4.15
Schedule 20	MYPKG → EGPSD → OMSLL → MYTPP → MYPKG	1200	3.96
Schedule 21	SGSIN → HKHKG → KRPUS → CNTAO → USLAX → CNSHA → TWKHH → MYTPP → SGSIN	1200	6.72
Schedule 22	MYPKG → MYTPP → TWKHH → CAVAN → AUBNE → MYPKG	1200	6.71
Schedule 23	NZAKL → AUBNE → NZAKL KRPUS → MYPKG → MYTPP → AUBNE → NZAKL	1200	1.17
Schedule 24	→ USLAX → CAVAN → JPYOK → TWKHH → HKHKG → CNTAO → KRPUS	1200	9.06
Schedule 25	USEWR → USCHS → GBFXT → DEBRV → ESALG → MAPTM → EGPSD → SAJED → USEWR	1200	8.15
Schedule 26	ITGIT → SAJED → ITGIT	1200	1.41
Schedule 27	MYTPP → CNYTN → OMSLL → SAJED → NGAPP → ZADUR → CNSHA → MYTPP	2400	10.54
Schedule 28	LKCMB → ZADUR → SAJED → AEJEA → LKCMB	2400	4.97
Schedule 29	AUBNE → NZAKL → TWKHH → CNYTN → SGSIN → AUBNE	2400	5.25
Schedule 30	MYTPP → SGSIN → MYTPP	2400	0.31

Table 7: Schedule output for World Small instance obtained by the LINERLIB solution (cont.).

Schedule	Vessel capacity	Number of vessels
Schedule 31 PABLB → USEWR → PAMIT → ESALG → NLRTM → USCHS → USMIA → CLSAI → PABLB	2400	8.92
Schedule 32 USCHS → PAMIT → USCHS	2400	1.11
Schedule 33 CLSAI → USLAX → CAVAN → PABLB → PAMIT → CLSAI	2400	6.04
Schedule 34 ITGIT → MAPTM → BRSSZ → GBFXT → DEBRV → SAJED → EGPSD → ITGIT	2400	7.36
Schedule 35 BEANR → DEBRV → DEHAM → ITGIT → MAPTM → BEANR	2400	2.79
Schedule 36 USMIA → NLRTM → BEANR → EGPSD → SAJED → AEJEA → NGAPP → PAMIT → USMIA	2400	9.99
Schedule 37 CNYTN → HKHKG → SGSIN → MYTPP → MYPKG → ZADUR → CNTAO → CNYTN	2400	6.68
Schedule 38 BRSSZ → PAMIT → USMIA → USCHS → ESALG → ITGIT → MAPTM → NGAPP → BRSSZ	2400	7.72
Schedule 39 ITGIT → AEJEA → MYTPP → OMSLL → SAJED → BEZEE → NLRTM → MAPTM → ITGIT	4200	7.83
Schedule 40 ESBCN → GBFXT → NLRTM → DEBRV → EGPSD → SAJED → ESBCN	4200	4.06
Schedule 41 LKCMB → SAJED → LKCMB	4200	2.30
Schedule 42 MYTPP → MYPKG → LKCMB → MYTPP	4200	1.67
Schedule 43 AEJEA → SAJED → ITGIT → ESALG → LKCMB → AEJEA	4200	5.19
Schedule 44 HKHKG → CNYTN → KRPUS → CAVAN → USLAX → JPYOK → MYTPP → HKHKG	4200	6.85
Schedule 45 LKCMB → MYTPP → LKCMB	4200	1.45
Schedule 46 OMSLL → SGSIN → MYTPP → HKHKG → CNSHA → MYPKG → AEJEA → EGPSD → ITGIT → OMSLL	4200	7.86
Schedule 47 GBFXT → NLRTM → ITGIT → GBFXT	4200	2.20
Schedule 48 HKHKG → SGSIN → MYPKG → AEJEA → SAJED → OMSLL → MYTPP → JPYOK → KRPUS → CNSHA → HKHKG	4200	7.48
Schedule 49 OMSLL → ESALG → DEBRV → SAJED → OMSLL	4200	4.57
Schedule 50 GBFXT → USCHS → CAMTR → DEBRV → NLRTM → GBFXT	7500	3.94
Schedule 51 CNYTN → ITGIT → HKHKG → CNYTN	7500	5.61

Table 8: Comparison of Vessel usage from our RL-based encoder-decoder approach and the LINERLIB solution for the World Small instance.

	RL-based	LINERLIB solution
Vessel capacity 450	24.11	23.43
Vessel capacity 800	39.10	27.37
Vessel capacity 1200	70.40	59.14
Vessel capacity 2400	79.19	71.68
Vessel capacity 4200	58.43	51.47
Vessel capacity 7500	11.07	9.55
Total vessel usage	282.30	242.64