Attention-based Reinforcement Learning for the Container Stowage Planning Problem

Mateusz Wilk (s4003748)

Supervisors:
Wei Liu & Dr. Yingjie Fan

Bachelor Thesis– Data Science and Artificial Intelligence

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

July 19, 2026

Abstract

Efficient yard operations are a critical bottleneck in maritime container terminals, where the order in which containers are retrieved for vessel loading directly determines the number of costly reshuffling operations incurred. While most existing work on container retrieval assumes a fully predetermined loading sequence, practical terminal operations often permit flexibility in the selection order, provided group-based vessel requirements are satisfied at each step. We formalize this setting as a Markov decision process and propose an attention-based deep reinforcement learning policy, which we term ACR-Policy, that learns to sequentially select containers from the yard under these partial ordering constraints. The model employs a transformer encoder–decoder architecture with a feasibility mask to restrict decisions to valid container choices at each step, and is trained using the REINFORCE algorithm with a greedy rollout baseline to stabilize learning. We further extend the problem to a multi-crane variant, in which parallel quay crane operations introduce an additional layer of scheduling complexity that places greater demands on the policy's capacity for long-term planning. Experiments across a range of single- and multi-crane scenarios show that ACR-Policy consistently outperforms MLP-based DQN and PPO baselines on medium and large problem instances, and substantially reduces solution times relative to a genetic algorithm on larger scenarios. On small instances, however, simpler baselines remain competitive. The model generalizes in a zero-shot fashion to larger unseen environments, though generalization quality is sensitive to the diversity of the training scenario.

Contents

1 Introduction

Global maritime trade is the backbone of international supply chains, responsible for over 80% of world merchandise trade by volume [26]. The scale of this system is substantial: seaborne cargo exceeded 12.3 billion tons in 2023 alone [27]. Container terminals are the physical nodes through which this trade flows, serving as the interface between ocean vessels and inland logistics networks [7]. The time a vessel spends at berth, its port stay, is a key operational cost driver, affecting both the terminal's throughput and the carrier's ability to maintain sailing schedules [22, 24]. Compressing port stay times therefore benefits all parties, and doing so depends critically on the efficiency of yard-side container handling.

A central source of inefficiency in yard operations is the reshuffling problem. Containers are stacked vertically in the yard and can only be accessed from the top of each stack [29, 35]. When a target container is buried beneath others, those blocking containers must be temporarily removed before retrieval can proceed, incurring additional crane movements known as reshuffles [25]. Since each reshuffle consumes time and equipment capacity, minimizing their total count over a loading sequence is an important operational objective. The difficulty of this task stems from its sequential and path-dependent nature: the choice of which container to retrieve at any given step alters the yard configuration and therefore affects how many reshuffles will be required for all subsequent steps.

The majority of existing research on this class of problems, often focus on the Container Relocation Problem (CRP), which assumes that the full retrieval order is fixed in advance [10, 11, 32, 31]. Under this assumption, the problem reduces to finding the most efficient way to execute a given sequence. In practice, however, vessel loading plans often specify various requirements such as weight constraints, destination port or access to electricity which can be abstracted using container groups. Our work focuses on allocating a fixed number of containers to a specific group rather than prescribing the loading sequence for all individual containers. This means that at each loading step, multiple yard containers may legitimately satisfy the requirement, and the algorithm has discretion over which one to select. Exploiting this flexibility intelligently can meaningfully reduce the total reshuffling cost, but doing so requires efficiently planning how each choice propagates through the remaining yard configuration.

We study this flexible selection setting and formulate it as a Markov decision process, in which a policy sequentially selects containers from the yard subject to group-based feasibility constraints. Alongside the standard single-crane formulation, we introduce a multi-crane extension in which several quay cranes operate in parallel across partitioned yard and vessel zones. Parallel crane operation, also explored in the context of container retrieval by [21], amplifies the planning challenge: the agent must simultaneously manage long-term yard accessibility and crane utilization, since poor allocation of cranes to containers can undermine even a well-reasoned retrieval sequence. As our experiments show, this added complexity tends to widen the performance gap between representationally richer and simpler policy architectures.

To address the decision-making challenges in both variants, we propose ACR-Policy, a deep reinforcement learning approach based on the transformer architecture [30]. The policy network uses an attention-based encoder to produce contextualized embeddings of all yard slots and the current vessel requirement, and a pointer-based decoder to produce a distribution over valid container choices. Training is performed using the REINFORCE algorithm [34] with a greedy rollout baseline [13], in which a frozen copy of the policy generates reference trajectories that are

periodically updated when the current policy demonstrates statistically significant improvement. This baseline strategy is shown to produce lower training variance than an Actor-Critic alternative [4], a finding consistent with recent observations on REINFORCE stability for combinatorial optimization [13, 14].

We compare ACR-Policy against three baselines: a genetic algorithm adapted from [9], a DQN agent [16], and a PPO agent [19], with the latter two using the MLP-based architecture proposed for container terminal operations by [8]. On medium and large single-crane scenarios, ACR-Policy matches or outperforms all baselines, while the genetic algorithm's single-threaded implementation makes it impractical at scale. In the multi-crane setting, ACR-Policy shows particularly strong relative gains on larger instances, though on small scenarios the MLP-based policies remain competitive or superior, suggesting a capacity-complexity trade-off that depends on problem scale. We also evaluate zero-shot generalization to environments larger than any seen during training, and find that while the architecture transfers reasonably well, the quality of generalization depends non-trivially on the diversity of the training scenario, where agents trained on overly small or homogeneous instances can degrade on large unseen variants.

1.1 Thesis overview

Section 2 discusses relevant literature and the limitations of existing experiments; Section 3 formalizes the problem definition and discusses problem constraints. Section 4 outlines the deep reinforcement learning approach used and discusses necessary preliminaries. The baseline algorithms are briefly described in Section 5. Section 6 describes the experimental setup and analyzes the results. Sections 7 and 8 reflect on the results and identify weaknesses in the experiment and potential directions for future work.

2 Literature review

The trade is currently dominated by lo-lo (lift on, lift off) ship designs [6], employing quay cranes to load and unload containers from the vessels. However, container stowage planning is an NP-hard problem [29], and decompositions such as the master planning problem and slot planning problem were proposed, where the former assigns the containers to general areas of the ship (bays), and the latter places containers into exact slots within a bay [29][35]. [17] used this two-phase approach to generate a near optimal plan for the entire problem.

While exact methods to find optimal solutions for MPP, have been developed [2][1], they were found to be unsuitable for larger problems and the use of heuristics was necessitated [2]. [28] used a deep reinforcement learning approach to solve a non-trivial master bay planning problem.

Methods to tackle the entire CSPP were developed. [3] used a genetic algorithm to solve the entire CSPP using representation by rules and simulation, [5] used RL with Proximal Policy Optimization [19], and achieved strong performance compared to other algorithms, while [36] used Monte Carlo Tree Search.

In order to solve the slot planning problem [20] used a variant of Deep Q-Learning [33][16], and showed a good generalization when evaluated on a larger set of containers than initially trained on.

[8] conducted a comparison experiment between DQN, QR-DQN, A2C, TRPO and PPO algorithms on the slot planning problem employing MLP-based policy networks optimized for this

problem.

Current research is lacking attention-based approaches applied specifically to the slot planning problem. However, the use of a transformer [30] as the policy network of the REINFORCE algorithm [34] was shown to perform comparably to highly optimized, problem-specific algorithms for other combinatorial optimization problems, such as the Travelling Salesman Problem [13]. It is even possible to train the algorithm once, and achieve robust performance on unseen, larger problem instances as it was done for the job shop scheduling problem by [15].

Recent studies focus on solving the Container Relocation Problem [32][21][31][11], which assumes a fixed loading sequence, using a similar attention architecture. Compared to SPP this is an easier problem to solve as the algorithm does not have to decide between seemingly equal containers, whereas SPP is a more flexible, where each decision affects the subsequent layout of the yard and therefore selection of future containers.

We focus our study on the REINFORCE algorithm as the nature of the container stowage problem involves a large discrete action space and sequential decision making [21]. However, REINFORCE can be unstable and careful baseline design is required to reduce variance and stabilize training [14][13][4].

3 Container Stowage Planning Problem

In container terminal operations, vessel stowage planning and yard operations are typically handled at different decision levels [22][1]. While vessel-level decisions such as slot allocation, stability constraints, and weight distribution are determined upstream, yard operations are responsible for retrieving containers to satisfy a given loading sequence. This hierarchical structure naturally decouples slot assignment from yard execution, yet introduces significant operational complexity due to the interaction between loading requirements and yard accessibility.

Motivated by this setting, we study a sequential container selection problem that captures the decision-making layer of yard operations under a predefined vessel loading plan.

Unlike the classical container retrieval problem (CRP) [10], where the retrieval order is fully specified, we consider a setting in which the loading sequence is only partially prescribed through group-based requirements. This introduces flexibility in the retrieval order, while retaining the stack-induced precedence constraints of the yard.

The yard is organized as a three-dimensional layout with B^Y bays, R^Y rows, and T^Y tiers. Let

$$S^Y = \{(b, r, \tau) : b \in \{1, \dots, B^Y\},\ r \in \{1, \dots, R^Y\},\ \tau \in \{1, \dots, T^Y\}\} \tag{1}$$

denote the set of yard slots, where each pair (b, r) defines a stack and containers are stored vertically along the tier dimension according to a last-in-first-out structure.

The vessel consists of a set of slots S^V, arranged in a predefined loading sequence

$$(v_1, v_2, \dots, v_m), \tag{2}$$

where m is the number of loading operations. Each vessel slot $v_t \in S^V$ is associated with a required group $g(v_t) \in \{1, \dots, G\}$. The group attribute abstracts multiple operational constraints, such as destination, container type, and compatibility requirements. In this formulation, feasibility is enforced through group matching.

Let C denote the set of containers initially stored in the yard. Each container $c \in C$ occupies exactly one yard slot and belongs to a group $g(c) \in \{1, \ldots, G\}$.

At each step $t = 1, \ldots, m$, a decision maker selects a container c_t from the current yard state to fill slot v_t. A container is feasible at step t if it is still present in the yard and satisfies the group requirement:

$$g(c) = g(v_t). \tag{3}$$

This group-based matching induces a partially flexible retrieval order, as multiple containers may be eligible for a given slot, and the choice among them affects future accessibility in the yard, Figure 1 illustrates an example layout of the yard where multiple containers are available for selection. The vessel slot has been assigned group 1, and containers in stack 1, 2 and 3 in the vessel with the same group are all available for selection.

Due to the stack structure, accessing a container may require temporarily removing containers stacked above it. Let X_t denote the yard state at step t, and let $\sigma_t(c)$ denote the number of containers stacked above container c in X_t. This quantity represents the reshuffling effort required to retrieve c.

In contrast to classical CRP formulations that explicitly model relocation decisions and destination stacks [9], we adopt an abstraction in which blocking containers are temporarily displaced and restored without altering their relative order, i.e. a blocking container from position (b, r, τ) will be reshuffled to $(b, r, \tau - 1)$, as a container from the bottom stack cannot be a blocking container.

This modeling choice isolates the selection layer of the problem, allowing us to focus on how container choices affect future accessibility, without the need for complex modeling of the relocation operations.

Let $C_t \subseteq C$ denote the set of containers remaining in the yard at step t. After selecting container c_t, the yard transitions from state X_t to X_{t+1} by removing c_t and updating the corresponding stack. The relative order of remaining containers within each stack is preserved.

The goal is to determine a feasible selection sequence

$$P = \langle c_1, c_2, \ldots, c_m \rangle \tag{4}$$

that minimizes the cumulative reshuffling cost:

$$\min_{P} \sum_{k=1}^{m} \sigma_t(c_t). \tag{5}$$

The proposed formulation generalizes the classical CRP by relaxing the assumption of a fully predetermined retrieval order. The resulting problem couples three interacting components: (i) group-based feasibility constraints, (ii) stack-induced precedence relations, and (iii) sequential decision-making. This interaction creates a nontrivial combinatorial structure in which each selection decision simultaneously determines immediate reshuffling cost and the future accessibility of remaining containers.

Despite the abstraction of relocation decisions, the problem retains the essential difficulty of stack-based retrieval systems, as decisions are path-dependent and locally optimal choices may lead to globally suboptimal outcomes. Consequently, the problem cannot be decomposed into independent per-step decisions and requires anticipating future states when selecting containers.

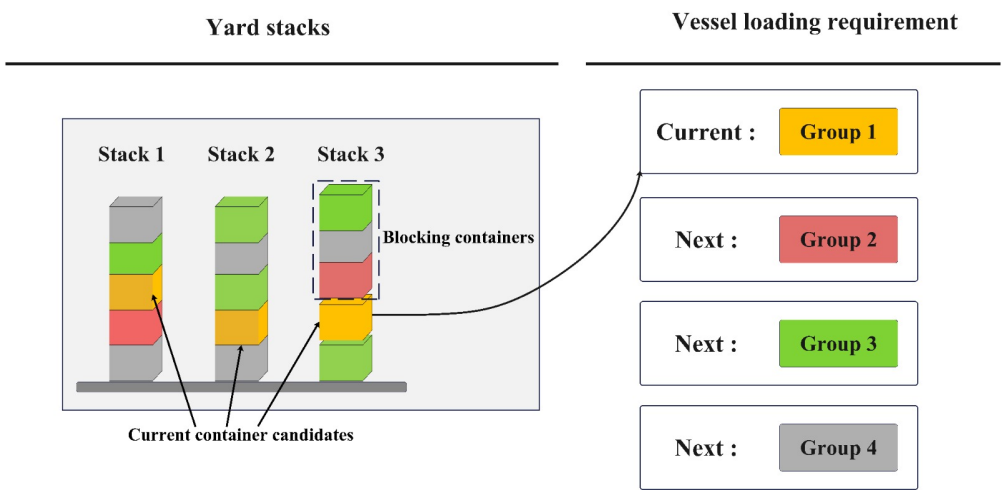


Figure 1: Illustration of the sequential container selection problem under group-based loading requirements. At each step, multiple feasible containers satisfy the current group constraint. Selecting different containers leads to different reshuffling costs and affects future accessibility.

3.1 Multi-crane variant

The environment can be augmented to employ mutliple quay cranes simultaneously to relocate containers in parallel. A single action in the environment represents choosing a (container, crane) pair. Consider a set of cranes $cr \in Cr$, the action space at each step is then $|C| \times |Cr|$. The yard and vessel bays are divided into $|Cr|$ zones.

Each container's stowage operation is assumed to be independent, and during environment initialization we create $|C|$ random values:

$$O = [o_1, o_2, ..., o_{|C|}], \tag{6}$$

corresponding to individual loading times for each container. Consider the total number of cranes $|Cr|$, a global clock t and a vector:

$$\phi = [\phi_1, \phi_2, ..., \phi_{|Cr|}] \tag{7}$$

The clock t tracks the time since the start of the stowage operation, while each ϕ_i represents the relative time after which crane cr_i is available again. When a crane cr_i is successfully selected to perform the stowage operation on container c_j, its time until available ϕ_i is advanced by the operation time of the selected container o_j. We can get the available cranes at each step by inspecting the non negative values of $\phi - t$ i.e. the cranes that are not busy at the current global time step. If there are no valid cranes at time t, it will be advanced to the earliest time when at least one crane cr_i is available.

This clock-based approach allows us to simulate parallel crane operations, while maintaining an underlying sequential processing structure which is simpler to model computationally.

This environment variant extends the problem formulation to minimizing the total reshuffle count and maximal utilization of all cranes for a multiple crane configuration.

4 Deep reinforcement learning approach

4.1 Preliminaries

A Markov Decision Process (MDP) is formally defined as a 6-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma, \mathcal{I})$. Where $\mathcal{S}$ denotes the set of states $s \in \mathcal{S}$, $\mathcal{A}$ the set of actions $a \in \mathcal{A}$, $\mathcal{T}$ the environment transition function, the agent observes the next state s_{t+1} after taking action a_t in state s_t, $\mathcal{T}(s_{t+1}|s_t, a_t)$, $\mathcal{R}$ the reward function - the agent recieves reward r_t when transitioning from state s_t to s_{t+1} after taking action a_t, $\mathcal{R}(s_t, a_t, s_{t+1}) = r_t$, $\gamma \in [0, 1]$ the discount factor which determines how the agent should weigh future rewards, and finally $\mathcal{I}(s)$ denotes the initial state distribution.

A reinforcement learning agent learns through trial and error by interacting with its environment [23]. It acts according to a policy $\pi(a_t|s_t)$, which specifies which action a_t to choose in state s_t. If the underlying distribution of $\mathcal{T}$ and $\mathcal{R}$ is known we can use Dynamic Programming to compute the value-function $V(s)$ or state-value function $Q(s, a)$ for an arbitrary state s and action a of the policy π:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi,\mathcal{T}}[\sum_{i=0}^{\infty} \gamma^i \times r_{t+1}|s_t = s, a_t = a] \tag{8}$$

$$V^{\pi}(s) = \mathbb{E}_{\pi,\mathcal{T}}[\sum_{i=0}^{\infty} \gamma^i \times r_{t+1}|s_t = s] \tag{9}$$

Reinforcement learning (RL) algorithms can be divided into two approaches, estimating the value-function $Q(s, a)$ of each state, known as value-based methods, or directly learning a policy $\pi(s, a)$ to choose action a in state s. Both approaches struggle to generalize to environments where storing a table $Q(s, a)$ or $\pi(s, a)$ for each possible $s \in \mathcal{S}, a \in \mathcal{A}$ would exceed the available memory. Therefore, approximation-based methods were devised, where the agent learns a parametrization θ where the number of parameters $|\theta| \ll |\mathcal{S}| \times |\mathcal{A}|$. It could then be applied to a parametrized policy $\pi_\theta(s, a)$ or a value function $Q_\theta(s, a)$, allowing us to extrapolate the important characteristics of the observed environment without the need to explicitly store every possible action and state transition.

A standard multilayer perceptron is a common choice for a policy or value network parametrization in Deep-RL; however, it struggles with learning representations for variable-input vectors, such as different yard and vessel sizes during training and evaluation. We motivate our choice of a transformer-based [30] policy network due to its permutation-invariant nature and the ability to generalize across different input sizes.

4.2 Problem formulation

We formulate the stowage process as an MDP and parameterize the policy using an attention-based transformer architecture. At each decision step s_t, the agent observes the yard configuration and the target vessel slot and then selects a yard slot from which a container is retrieved and loaded onto the ship. A solution is defined as a sequence of actions

$$\mathbf{a} = (a_1, a_2, \ldots, a_T), \tag{10}$$

where each action a_t corresponds to selecting a yard slot.

Our model defines a stochastic policy $\pi_\theta(\mathbf{a} \mid s_1)$ to generate a sequence of actions given an initial state s_1, given a policy parametrization θ. The policy is factorized as:

$$\pi_\theta(\mathbf{a} \mid s_1) = \prod_{t=1}^{T} \pi_\theta(a_t \mid s_t), \tag{11}$$

where s_t denotes the state of the environment at the decision step t, and s_{t+1} is obtained from the environment transition after executing action a_t. The policy network follows an encoder–decoder structure. The encoder maps the current state to contextualized embeddings. At each decision step, the decoder produces a distribution over feasible actions conditioned on the encoder output. A feasibility mask is applied to restrict the action space, so that the actions are sampled only from valid yard slots that are occupied and whose group matches the target vessel slot.

4.3 Encoder

At the decision step t, the state observed by the agent is represented by all yard slots together with the target vessel slot. Let the encoder input be

$$X_t = [x_1, x_2, \ldots, x_m, x_c], \tag{12}$$

where $x_i \in \mathbb{R}^{d_x}$ denotes the feature vector of yard slot i, and $x_c \in \mathbb{R}^{d_x}$ denotes the feature vector of the current target vessel slot. In our implementation, $d_x = 5$, corresponding to the slot attributes (bay, row, tier, occupied, group).

Since the input is represented as a set of slots rather than a sequence and each token already contains explicit spatial attributes such as bay, row, and tier, we do not use positional encodings. From the d_x-dimensional input features x_i, the encoder computes the initial d-dimensional embeddings $h_i^{(0)}$ (we use $d = 128$) through a learned linear projection:

$$h_i^{(0)} = W^x \tilde{x}_i + b^x \tag{13}$$

where $W^x \in \mathbb{R}^{d \times d_x}$ is a learnable parameter.

The embeddings are updated using N stacked attention layers. Let $H^{(\ell-1)} \in \mathbb{R}^{(m+1)\times d}$ denote the embedding matrix entering the layer $\ell \in \{1, \ldots, N\}$. Each layer consists of two sublayers: a multi-head attention (MHA) sublayer that enables information exchange between tokens, and a position-wise feed-forward (FF) sublayer. Each sublayer is equipped with a residual connection and layer normalization:

$$\hat{H}^{(\ell)} = \mathrm{LN}\Big(H^{(\ell-1)} + \mathrm{MHA}^{(\ell)}(H^{(\ell-1)})\Big), \tag{14}$$

$$H^{(\ell)} = \mathrm{LN}\Big(\hat{H}^{(\ell)} + \mathrm{FF}^{(\ell)}(\hat{H}^{(\ell)})\Big). \tag{15}$$

Each attention layer uses 4 heads, and the feed forward sublayer has a hidden dimension of 256 with ReLU activation. After N layers, the encoder produces the final embeddings $H_t \in \mathbb{R}^{(m+1)\times d}$, which represent the encoded state. The embeddings corresponding to yard slots are used by the decoder as candidate actions, while the embedding of the target vessel slot provides contextual information.

4.4 Decoder

At each decision step, the decoder produces a distribution over feasible actions using a pointer mechanism. Unlike autoregressive decoders that explicitly condition on previously selected actions [13], our approach relies on the environment state to encode all relevant historical information.

The decoder applies a cross-attention operation using a learnable query vector q_0 as the query and the encoder output H_t as keys and values:

$$c_t = \mathrm{MHA}(q_0, H_t, H_t). \tag{16}$$

A residual connection followed by layer normalization is then applied:

$$\tilde{q}_t = \mathrm{LN}(q_0 + c_t). \tag{17}$$

The resulting vector $\tilde{q}_t$ serves as a context-aware current decision state.

The decoder computes a compatibility score between $\tilde{q}_t$ and each candidate action. Let h_i denote the encoded embedding associated with action i. The query and action embeddings are projected as

$$q_t^p = W_p^Q \tilde{q}_t, \tag{18}$$

$$k_i^p = W_p^K h_i. \tag{19}$$

The unnormalized compatibility score is then given by the scaled dot product

$$u_i = \frac{q_t^p (k_i^p)^\top}{\sqrt{d}}. \tag{20}$$

To improve numerical stability and prevent overly large logits, the scores are clipped using a scaled hyperbolic tangent:

$$\bar{u}_i = C \tanh(u_i), \tag{21}$$

where C is a constant and is set to 10 in our implementation.

A feasibility mask $M_t(i) \in \{0, 1\}$ is then applied to enforce environment constraints:

$$\tilde{u}_i = \begin{cases} \bar{u}_i, & M_t(i) = 0, \\ -\infty, & M_t(i) = 1. \end{cases} \tag{22}$$

In our environment, the mask excludes invalid yard slots, so that actions are sampled only from slots that are occupied and whose group matches the target vessel slot.

Finally, the masked scores are normalized using a softmax function to obtain the action distribution

$$\pi_\theta(a_t = i \mid s_t) = \frac{\exp(\tilde{u}_i)}{\sum_j \exp(\tilde{u}_j)}. \tag{23}$$

During training, actions are sampled from this distribution to encourage exploration. During evaluation, the action with the highest probability is selected:

$$\pi_\theta(a_t = i \mid s_t) = argmax_i \frac{\exp(\tilde{u}_i)}{\sum_j \exp(\tilde{u}_j)}. \tag{24}$$

Compared to sequence-based decoding approaches [13][4][14], this design avoids maintaining an explicit autoregressive state. Instead, all historical information is encoded in the environment state and reprocessed by the encoder at each decision step, which simplifies the decoder while preserving the full decision context.

4.5 Policy Optimization via REINFORCE

We train the policy using the REINFORCE algorithm [34]. This choice is motivated by two characteristics of our problem. First, the decision at each step is made over a large discrete action space of yard slots, for which policy-gradient methods are naturally applicable. Second, the quality of a retrieval action depends primarily on its effect on the full future retrieval sequence, rather than on an immediate local reward.

Recent studies [21] have noted that REINFORCE is starting to be widely adopted for Combinatorial Optimization problems, as it does not suffer from instability in large discrete action spaces like DQN, and PPO due to the action-value approximation errors for the former and sensitivity of advantage estimation for the latter.

Let $\tau = (s_1, a_1, \ldots, s_T, a_T)$ denote a trajectory generated by the policy. For each action a_t, let ρ_t denote the number of reshuffling operations induced by retrieving the selected container. The episodic return is defined as the negative total reshuffling cost:

$$\mathcal{R}(\tau) = -\sum_{t=1}^{T} \rho_t. \tag{25}$$

In our environment, intermediate rewards are set to zero and the terminal reward equals $R(\tau)$. With discount factor $\gamma = 1$, the return from each decision step is therefore equal to the final episodic return. This sparse reward design is motivated by the sequential nature of the problem: the impact of a container selection is not limited to the immediate reshuffling cost, but also affects the future accessibility of remaining containers. Providing rewards only at the end of the episode encourages the policy to optimize the overall retrieval sequence rather than making purely myopic decisions.

The training objective is to maximize the expected return:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]. \tag{26}$$

Its gradient is estimated using REINFORCE with a baseline:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T} A_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right], \tag{27}$$

where A_t is the advantage term.

To reduce variance, we use a greedy rollout baseline [13]. Specifically, for each batch of training instances, the current stochastic policy generates sampled trajectories, while a frozen baseline policy generates greedy trajectories on the same instances. Let τ^{sample} and τ^{base} denote the sampled and baseline trajectories, respectively. The advantage is defined as

$$A_t = G_t^{\text{sample}} - G_t^{\text{base}}, \tag{28}$$

where G_t^{sample} and G_t^{base} are the returns from step t onward for the sampled and baseline trajectories. Because the reward is terminal-only and $\gamma = 1$, this reduces to comparing the final episodic returns of the two trajectories.

To encourage exploration, we add an entropy regularization term. The policy parameters are updated by minimizing the following loss:

$$L(\theta) = -\mathbb{E}\left[\sum_{t=1}^{T} A_t \log \pi_\theta(a_t \mid s_t)\right] - \beta\, \mathbb{E}\left[\sum_{t=1}^{T} \mathcal{H}(\pi_\theta(\cdot \mid s_t))\right], \tag{29}$$

where β is the entropy coefficient and $\mathcal{H}(\cdot)$ denotes policy entropy. We optimize this objective using Adam [12].

Algorithm 1 REINFORCE with Greedy Rollout Baseline, adapted from [13]

Input: number of epochs E, steps per rollout T, number of parallel environments $\mathcal{E}$, significance α, entropy coefficient β
Init θ, $\theta^{\text{base}} \leftarrow \theta$
$s_i \leftarrow \text{InitializeEnvironment}() \ \forall i \in \{1, \dots, \mathcal{E}\}$
for epoch $= 1, \dots, E$ **do**
  $G_t^{\text{sample}} \leftarrow \text{SampleRollout}(s_i, \pi_\theta) \ \forall i \in \{1, \dots, \mathcal{E}\}$
  $G_t^{\text{base}} \leftarrow \text{GreedyRollout}(s_i, \pi_{\theta^{\text{base}}}) \ \forall i \in \{1, \dots, \mathcal{E}\}$
  $A_t = G_t^{\text{sample}} - G_t^{\text{base}}$
  $\mathcal{L}(\theta) = -\mathbb{E}\left[\sum_{t=1}^{T} A_t \log \pi_\theta(a_t \mid s_t)\right] - \beta\, \mathbb{E}\left[\sum_{t=1}^{T} \mathcal{H}(\pi_\theta(\cdot \mid s_t))\right]$
  $\theta \leftarrow \text{Adam}(\theta, \nabla\mathcal{L}(\theta))$
  if $\text{OneSidedPairedTTest}(\pi_\theta, \pi_{\theta^{\text{base}}}) < \alpha$ **then**
    $\theta^{\text{base}} \leftarrow \theta$
  end if
end for=0

As visualized in Algorithm 1 adapted from [13], the baseline policy is initialized as a copy of the current policy and kept fixed during training. After each training iteration, we compare the episodic returns of the current policy and the baseline policy on matched training instances using a one-sided paired t-test, where the baseline update p-value is treated as a hyperparameter. The baseline is updated only when the current policy significantly outperforms the existing baseline. This update rule stabilizes training by preventing noisy or premature baseline replacement.

4.6 Value Network baseline

It is also possible to explore alternative baseline representations, such as a learned value network baseline, which changes the REINFORCE Algorithm to Actor-Critic. The advantage is then defined as:

$$A_t = G_t^{\text{sample}} - \theta_{critic}(\tau), \tag{30}$$

Where $\theta_{critic}(\tau)$ defines the output of the value network given observation τ. This baseline can adapt to specific environment situations, giving unbiased estimates of the actions chosen by the policy network. The value network uses the same encoder architecture as the policy network, however, instead of a pointer-based decoder, it uses a single attention head to reduce the dimensionality of the encoder output:

$$c_t = \text{MHA}(q_0, H_t, H_t). \tag{31}$$

Where H_t denotes the encoder output and q_0 a learned start embedding, and then a number of feed-forward layers $\ell \in \{1, \dots, N\}$ to classify the pooled embeddings into a single scalar value:

$$\hat{V}_s = \text{FFN}^{\ell-1}(c_t), \tag{32}$$

$$V_s = \text{FFN}^{\ell}(\hat{V}_s). \tag{33}$$

Where v_s denotes the final output of the critic θ_{critic} i.e. the estimated state value. This baseline design can stabilize training performance compared to a vanilla REINFORCE implementation, however, it is potentially prone to instability while simultaneously training the policy and value networks.

5 Baselines

We compare our attention architecture to MLP-based implementations of Deep Q-Learning [16] and Proximal Policy Optimization [19]. Recent research has focused on employing genetic algorithms to solve CRP and related problem variants [9], therefore we also include a genetic algorithm implementation to solve SPP.

5.1 Deep Q-Learning

Algorithm 2 DQN with Experience Replay and Action Masking, adapted from [16]

Input: total timesteps T, number of parallel environments $\mathcal{E}$, discount factor γ, soft update rate τ, target update frequency f_{target}, warmup steps T_{warm}, ε-decay schedule $\varepsilon(\cdot)$
Init θ, $\theta_{old} \leftarrow \theta$, replay buffer $\mathcal{D} \leftarrow \emptyset$
$s_1 \leftarrow$ InitializeEnvironment()
for $t = 1, \ldots, T$ **do**
  Compute valid-action mask M_t from s_t
  With probability $\varepsilon(t)$ select random $a_t \in M_t$, otherwise $a_t = \arg\max_{a \in M_t} Q_\theta(s_t, a)$
  Execute a_t, observe r_t, s_{t+1}, M_{t+1}; store $(s_t, a_t, r_t, s_{t+1}, M_{t+1})$ in $\mathcal{D}$
  if $t \geq T_{\text{warm}}$ **then**
    Sample minibatch $\{(s_j, a_j, r_j, s_{j+1}, M_{j+1})\}_{j=1}^{B}$ from $\mathcal{D}$
    Compute TD targets using the target network with action masking:
    $y_j = r_j + \gamma \max_{a \in M_{j+1}} Q_{\theta_{old}}(s_{j+1}, a)$
    $\mathcal{L}(\theta) = \mathbb{E}\left(y_j - Q_\theta(s_j, a_j)\right)^2$
    $\theta \leftarrow \text{Adam}(\theta, \nabla\mathcal{L}(\theta))$
  end if
  Perform periodic soft update of θ, according to τ, to improve stability:
  if $t \bmod f_{\text{target}} = 0$ **then**
    $\theta_{old} \leftarrow \tau\,\theta + (1 - \tau)\,\theta_{old}$
  end if
end for=0

Deep Q-learning (DQN) [16] is a value-function approximation algorithm derived from Q-learning [23], which is an off-policy TD algorithm, meaning it learns a different target policy from its behavioral policy i.e. it can learn to select optimal actions even if the behavioral policy keeps exploring. We model the behavioral policy using an ϵ-greedy strategy for selecting action a_t:

$$a_t = \begin{cases} \text{random action } a_t, & \text{with probability } \epsilon, \\ argmax_{a \in M_t} Q_\theta(s_t, a), & \text{otherwise.} \end{cases} \tag{34}$$

Value-based methods attempt to learn the optimal value function $Q^*(s, a)$ for every state-action pair $s \in \mathcal{S}, a \in \mathcal{A}$. Since Q-learning cannot generalize state representations as it simply stores a table for each state and action of size $|\mathcal{S}| \times |\mathcal{A}|$, it is prone to memory limitations. Deep Q-learning aims to parametrize the Q-value function according to some set of parameters θ. We use a multi-layer perceptron with two hidden layers and a dimension of 64 as θ.

To maintain stability we perform periodic updates of θ, saving intermediate parameters as θ_{old}, which is used for generating episodes and estimating targets, where r_j denotes a sampled reward from buffer $\mathcal{D}$:

$$y_j = r_j + \gamma \max_{a \in M_{j+1}} Q_{\theta_{old}}(s_{j+1}, a) \tag{35}$$

Given the target y_j we can calculate the loss using MSE:

$$\mathcal{L}(\theta) = \mathbb{E}\left(y_j - Q_\theta(s_j, a_j)\right)^2, \tag{36}$$

which is optimized using gradient descent and Adam [12].

The pseudo code for this algorithm is visualized in algorithm 2. We first compute the action mask for each state, and take a random action according the ϵ-greedy action selection, storing the transition in a replay buffer $\mathcal{D}$. After we store a sufficient amount of samples in the buffer, where the global timestep t exceeds T_{warm} we can start sampling minibatches of previous transitions and optimizing the network θ. To maintain stability, we perform a soft update of the network used for sampling actions every f_{target} steps:

$$\theta_{old} \leftarrow \tau\,\theta + (1-\tau)\,\theta_{old} \tag{37}$$

where τ is a hyperparameter controlling the proportion of the update.

5.2 Proximal Policy Optimization

Trust Region Policy Optimization A clipped surrogate objective is defined as follows [18]:

$$\underset{\theta}{\text{maximize}}\ \mathbb{E}\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A_t \tag{38}$$

$$\text{subject to: } \mathbb{E}[\text{KL}[\pi_{old}(.|s_t)\pi(.|s_t)]] \leq \delta \tag{39}$$

Where θ_{old} denotes the policy parameters before the update. TRPO methods estimate the the solution to this problem using a conjugate gradient algorithm by making a linear approximation of the objective, shown in Equation 38 and a quadratic approximation of the constraint, which is shown in Equation 39.

PPO, Actor-Critic style We follow an Actor-Critic style Proximal Policy Optimization algorithm [19], employing a single-hidden layer MLP of size 64 for both the policy and value networks.

In comparison to policy-gradient methods it attemps to normalize the size of the probability ratio $\mathbb{E}\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ of the surrogate objective by clipping it within an interval $[1-\epsilon, 1+\epsilon]$, where ϵ denotes the magnitude of the update.

The objective is to improve training stability, as a large update magnitude caused by a policy gradient update (such as the update rule shown in Equation 29) can lead to destructively large policy updates.

Algorithm 3 PPO with GAE

Input: number of iterations I, steps per rollout T, number of environments $\mathcal{E}$, mini-batch size B, update epochs K, clipping coefficient ε, entropy coefficient β, value coefficient c_v
Init actor-critic parameters θ
$s_i \leftarrow \text{InitializeEnvironment}() \;\; \forall i \in \{1, \ldots, \mathcal{E}\}$
for iteration $= 1, \ldots, I$ **do**
  $\text{SampleRollout}(s_i, \pi_\theta) \;\; \forall i \in \{1, \ldots, \mathcal{E}\}$
  for epoch $= 1, \ldots, K$ **do**
    for each mini-batch of size B sampled from the $\mathcal{E}T$ rollout transitions **do**
      Compute and normalize advantage estimates $\hat{A}$ using GAE
      Compute returns $\hat{R}_t = \hat{A}_t + V_\theta(s_t)$
      $\mathcal{L}^{\text{clip}} = -\mathbb{E}\Big[\min\Big(\rho\,\hat{A},\; \text{clip}(\rho, 1 \pm \varepsilon)\,\hat{A}\Big)\Big]$
      $\mathcal{L}^{V} = \frac{1}{2}\mathbb{E}\Big[\Big(V_\theta(s) - \hat{R}\Big)^2\Big]$
      $\mathcal{L}(\theta) = \mathcal{L}^{\text{clip}} + c_v\mathcal{L}^{V} - \beta\,\mathbb{E}[\mathcal{H}(\pi_\theta(\cdot \mid s))]$
      $\theta \leftarrow \text{Adam}(\theta, \nabla\mathcal{L}(\theta))$
    end for
  end for
end for=0

The loss is computed as a factor of the value-function loss, i.e. the MSE between the returns and an estimated state value using the critic network:

$$\mathcal{L}^{V} = \frac{1}{2}\mathbb{E}\Big[\Big(V_\theta(s) - \hat{R}\Big)^2\Big] \tag{40}$$

where $\mathcal{L}^{V}$ denotes the value function loss, and the clipped loss $\mathcal{L}^{\text{clip}}$ of the surrogate objective:

$$\mathcal{L}^{\text{clip}} = -\mathbb{E}\Big[\min\Big(\rho\,\hat{A},\; \text{clip}(\rho, 1 \pm \varepsilon)\,\hat{A}\Big)\Big] \tag{41}$$

The final loss $\mathcal{L}(\theta)$ becomes:

$$\mathcal{L}(\theta) = \mathcal{L}^{\text{clip}} + c_v\mathcal{L}^{V} - \beta\,\mathbb{E}[\mathcal{H}(\pi_\theta(\cdot \mid s))] \tag{42}$$

Where c_v denotes the value coefficient. We augment the loss by the entropy regularization term $-\beta\,\mathbb{E}[\mathcal{H}(\pi_\theta(\cdot \mid s))$ to encourage exploration.

Algorithm 3 shows the pseudo code for a generic PPO implementation. The algorithm first performs transitions according to actions selected by the policy π_θ, after which it samples a minibatch of these transitions and performs gradient descent using Adam[12], according to the loss from Equation 42.

5.3 Genetic Algorithm

A genetic algorithm (GA) adapted from [9] was shown to perform consistently well in the current environment. The main components of this algorithm are: two cross-bit mutation, two cross-bit crossover, and fitness evaluation.

We use a chromosome to store the order of containers for the entire episode, with its length equal to the number of containers to stow. Considering a scenario with 20 containers to stow, an example chromosome could be:

```
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20]
```

Algorithm 4 Genetic Algorithm with Tournament Selection, adapted from [9]

Input: number of generations G, population size N, Number of containers C, mutation rate μ, tournament selection sample size α
P = InitPopulation() of N chromosomes with randomly shuffled values from $1, ..., C$
for generation $g = 1, ..., G$ **do**
    O = TwoCrossbitCrossover(P)
    Mutate(O, μ)
    F_P, F_O = EvaluateFitness(P), EvaluateFitness(O)
    P = TournamentSelection(O, P, F_O, F_P, α) of N best individuals
end for=0

Algorithm 4 shows the general procedure of a GA, we first initialize a population of size N, after which we repeatedly apply crossover to create offsprings, we mutate a proportion of size μ of the off-springs and select the best N individuals to become the new population according to the $EvaluateFitness()$ function. The procedure is repeated for G generations.

Two cross-bit crossover The crossover is performed as follows: select two points $a \in [1, \frac{n}{2}]$ and $b \in [\frac{n}{2}+1, n]$ where n is the length of the chromosome $n = |C|$ where C is the set of all containers in the yard. Child 1 inherits points from $[1, a]$ and $[b, n]$ from parent 1 directly, where child 2 inherits points from $[1, a]$ and $[b, n]$ from parent 2. The points within $[a, b]$ are inherited in the relative order in which they are on the entire chromosome of the other parent, i.e, parent 2 for child 1 and parent 1 for child 2.

For example consider two parents 1 and 2:

```
(1) [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20]
(2) [4 5 18 19 8 7 6 13 16 11 10 20 3 14 15 9 17 1 2 12]
```

And children 1 and 2 for $a = a, b = 16$:

```
                 a                                        b
(1) [1 2 3  4 | 5 8 7 6 13 16 11 10 14 15 9 12 | 17 18 19 20]
(2) [4 5 18 19| 3 6 7 8 9 10 11 13 14 15 16 20 | 17 1  2  12]
```

For both children the first and last four values are copied directly from their outer parent. The inner values are preserved in the relative order of the remaining values in the inner parent.

Two cross-bit mutation Considering randomly generated points $a \in [1, \frac{n}{2}]$ and $b \in [\frac{n}{2}+1, n]$ for each offspring chosen according to a mutation rate μ we exchange the values at points a and b in the chromosome with each other. This procedure reduces the randomness of the mutation and is naturally applicable for our environment setting.

Fitness evaluation The order of containers in the chromosome is treated as a priority list instead of a fixed order of selection. The fitness of each candidate is evaluated by selecting the first valid action from the chromosome, starting from the left side. We run simulated rollouts in the environment, treating this priority ordering as a policy, repeatedly selecting the first valid container from the chromosome until terminating the episode. The final reshuffle count is treated as the fitness of the candidate. This procedure allows us to abstract the finer details during crossover such as validity of the entire action sequence, as we can simply rely on the action mask provided by the environment during the rollout, while still preserving potential of strong selection orders for later generations of candidates.

Selection We employ a tournament selection procedure, where for N parents and N off-springs, creating a combined population of size $2N$, we select N individuals with the highest fitness compared to a random sample of size α for each individual. This ensures that the best performing individuals have a chance to survive to the next generation.

6 Experiment

All reinforcement learning algorithms were evaluated over 50000 timesteps in each environment scenario, and their results were averaged over 30 repetitions. The algorithms were tested on an evaluation environment instance after every training epoch. We used seeding for reproducibility. We used the *Optuna* framework, with 60 trials for each algorithm-environment pair, using a single representative scenario for the single-crane and the multi-crane experiment to tune the agents on. The hyperparameter optimization results are shown in the Section 9 (Appendix), in Table 4 and 5.

The experiments were run on a server with two Nvidia RTX 3090 GPUs with 24GBs of VRAM each. The CPU used was a 24-core/48-thread Intel Xeon Silver 4214, with 256GBs of RAM.

6.1 Environment configuration

Scenario	Vessel Dimension	Yard Dimension	Containers to Stow	Group Types	Cranes
S1	3×5×3	3×5×3	45	3	1
S2	3×5×3	8×5×5	45	8	1
S3	8×5×5	8×5×5	200	3	1
S4	8×5×5	8×5×5	200	8	1
S5	10×8×8	10×8×8	400	8	1
S6	10×8×8	10×8×8	400	16	1
M1	3×5×3	3×5×3	45	3	3
M2	3×5×3	8×5×5	45	8	3
M3	8×5×5	8×5×5	200	3	5
M4	8×5×5	8×5×5	200	8	5
M5	10×8×8	10×8×8	400	8	5
M6	10×8×8	10×8×8	400	16	5

Table 1: Experimental environment configurations across different scenarios. They range from small (3×5×3), medium (8×5×5) and large (10×8×8).

We tested the algorithms on environments of increasing complexity, including larger vessel and yard sizes, a higher number of containers to stow, as well as more groups. Table 1 summarizes the individual experimental scenarios. They range from small (scenarios 1, 2), medium (scenarios 3, 4), and large (scenarios 5, 6). S4 and M4 were chosen as the scenario to tune the hyperparameters for all deep reinforcement learning algorithms, for the single-crane and multi-crane variants, respectively.

Scenario		DQN	PPO	Genetic	ACR-Policy			
					Train S1	Train S2	Train S3	Train S4
S1	Res.	**7.00**	**7.00**	**7.00**	**7.00**	**7.00**	**7.00**	**7.00**
	Time	229.51	41.70	73.75	68.55	76.11	72.56	75.54
S2	Res.	45.33	44.07	**41.23**	44.63	43.70	44.17	44.93
	Time	64.17	47.10	86.71	70.07	78.08	74.49	81.67
S3	Res.	132.70	133.17	131.00	**129.10**	129.70	129.23	129.37
	Time	64.24	44.38	377.23	68.30	75.70	75.91	83.26
S4	Res.	184.33	186.13	183.27	**183.00**	**183.00**	**183.00**	**183.00**
	Time	63.69	45.62	416.24	68.35	77.78	75.10	84.31

Table 2: Comparison of REINFORCE with greedy baseline, DQN, PPO and a genetic algorithm on the final mean reshuffles across small and medium single-crane evaluation scenarios. Values represent mean reshuffles and mean runtime in seconds over 30 repetitions.

6.2 Comparison with Baseline Methods

We now report the results of comparing our proposed Attention-based Container Retrieval policy (ACR-Policy) with other algorithms.

The experiment compared our ACR-Policy against three baseline methods: a genetic algorithm (GA) adapted from [9], a Deep Q-Learning (DQN) agent [16], and a Proximal Policy Optimization (PPO) agent [19]. For both DQN and PPO, we adopt the neural network architecture proposed by [8], which was originally developed for container terminal operations. The experiment serves two main purposes. First, it compares our model with previous non-attention-based methods. Second, it investigates the ability of the proposed representation to generalize across problem sizes by assessing whether a model trained on one scenario can be effectively applied to scenarios of different scales.

Single-crane variant Table 2 summarizes these results for the small (S1, S2) and medium (S3, S4) single-crane scenarios. The best value for each scenario is highlighted in bold font. It is apparent that the GA baseline takes a similar amount of time to generate a solution as all other Deep RL methods for scenarios S1, and S2, but due to the single-threaded nature of it's implementation, it scales poorly to larger problem instances in S3 and S4. The GA can find the best solution on average in scenarios S1 and S2, where in S1, all models find equally good solutions of 7 reshuffles, in S2 it even outperforms the second best algorithm, ACR-Policy trained on S2, by over 5%. PPO and DQN produce the worst solutions in all scenarios apart from S1, outperformed by both GA and ACR-Policy. Only in S1 can they find the best solution, while in S2 PPO outperforms the ACR-Policy trained on S1, S3, and S4.

While in S2, PPO seems to perform better out of the two algorithms, it is outperformed by DQN in larger problem instances, namely S3 and S4. However, DQN is still 2.7% behind the optimal solution generated by the proposed ACR-Policy in S3 and 0.7% in S4. The proposed ACR-Policy falls behind the baselines in S2, showing performance comparable to PPO, but excels in larger

problem instances. In both S3 and S4, the reinforce variants can find the best known solutions out of all the algorithms compared. Interestingly, all ACR-Policy variants trained on all of the four scenarios perform very similarly on each evaluation scenario. The small variation between the ACR-Policy variants could be due to the unequal number of degrees in the data available, potentially preventing the model from interpolating finer details for less verbose training scenarios.

Considering the number of groups, for instance, if the model is only trained on a scenario with 3 groups, namely S1 and S3, it might find it more difficult to act on an environment with 8 groups such as S2 and S4, due to the lack of diversity in training data. However, as shown in the results of Table 2, these effects play a very minor role in the performance of the final model, as both ACR-Policy variants trained on S1 and S3, as well as S2 and S4 exhibit similar final performance, likely due to the effective feature normalization and generalization capability of the transformer architecture.

Scenario		**DQN**	**PPO**	**Genetic**	**ACR-Policy**			
					Train M1	Train M2	Train M3	Train M4
M1	Res.	0.13	**0.00**	1.60	1.50	2.77	2.03	2.73
	Time	70.14	88.35	143.39	95.28	99.07	103.39	103.12
M2	Res.	17.80	**14.63**	18.69	22.10	19.57	21.43	19.97
	Time	75.79	87.89	147.24	97.76	100.48	93.28	96.07
M3	Res.	12.07	9.57	83.55	2.40	1.27	1.27	**0.97**
	Time	66.99	76.41	766.11	102.95	180.74	254.07	105.14
M4	Res.	106.47	109.23	120.36	96.73	96.13	96.50	**95.03**
	Time	72.16	82.72	769.02	93.30	97.36	248.17	169.43

Table 3: Comparison of REINFORCE with greedy baseline, DQN, PPO and a genetic algorithm on the final mean reshuffles across small and medium multi-crane evaluation scenarios. Values represent mean reshuffles and mean runtime in seconds over 30 repetitions.

Multi-crane variant The results for small (M1, M2) and medium (M3, M4) multi-crane scenarios are shown in table 3. Bold font highlights the lowest reshuffle count for each scenario.

The Genetic algorithm produces reasonable solutions in scenarios M1 and M2, staying within 2 and 5 shifters of the optimal solution in M1, and M2 respectively, which is similar to the performance of the best ACR-Policy variants on these scenarios. However, it creates the worst-performing solutions in M3 and M4 out of all algorithms tested, where in M3 its solution is two orders of magnitude worse than the best solution. Similarly to the results from Table 2, the scalability of this algorithm to larger problem sizes is heavily limited by its serial implementation. The running time for scenarios M3 and M4 is approximately 10 times higher than most other algorithms tested.

PPO and DQN create reasonably good solutions in all scenarios, where especially in smaller problem instances such as M1 and M2 they find some of the best solutions. PPO in particular shows very competitive performance in M1, finding on average a solution with 0 mean reshuffles, followed by DQN with 0.13 reshuffles. The algorithms start to exhibit limitations of their architectures in

M3 and M4, where the best solution found by ACR-Policy is between 9 and 13 reshuffles lower. As both of these models use relatively small policy and value networks, they might approach a limitation with the capacity of the model. However, DQN and PPO are prone to instability during training as noted by [21], therefore, it is plausible that larger environments might amplify these weaknesses.

ACR-Policy variants trained on M2, M3 and M4 perform worse than all baselines in scenarions M1 and M2, while the variant trained on M1 slightly outperforms the GA implementation. In M2 the average optimality gap of the four ACR-Policy variants with PPO is 41.9%. This architecture, however, demonstrates the strongest performance in more complex problem instances, in M3 the ACR-Policy variant trained on scenario M4 finds solutions (mean reshuffles 0.97) that are an order of magnitude better than the best performing baseline, PPO (9.57 mean reshuffles). Similarly, in M4 the best solution is found by the same variant, where the gap to two Deep RL baselines DQN and PPO is above 11 reshuffles. It is also important to note that the ACR-Policy variants do not show a significant difference in performance when choosing different training scenarios.

The multi-crane variant demonstrates better scalability of the ACR-Policy architecture to larger environments than the single-crane results, however it also identifies its weaknesses when evaluating on smaller problem instances, where the agent could overfit due to higher model capacity. As the multi-crane environment is a more difficult problem variant, the algorithms need to learn how to effectively utilize all cranes in conjunction with long-term planning of the container layout in the yard to minimize reshuffle counts. It is plausible that the standard MLP-based policy and value networks used by DQN and PPO lack the capacity to perform effectively in larger problem instances.

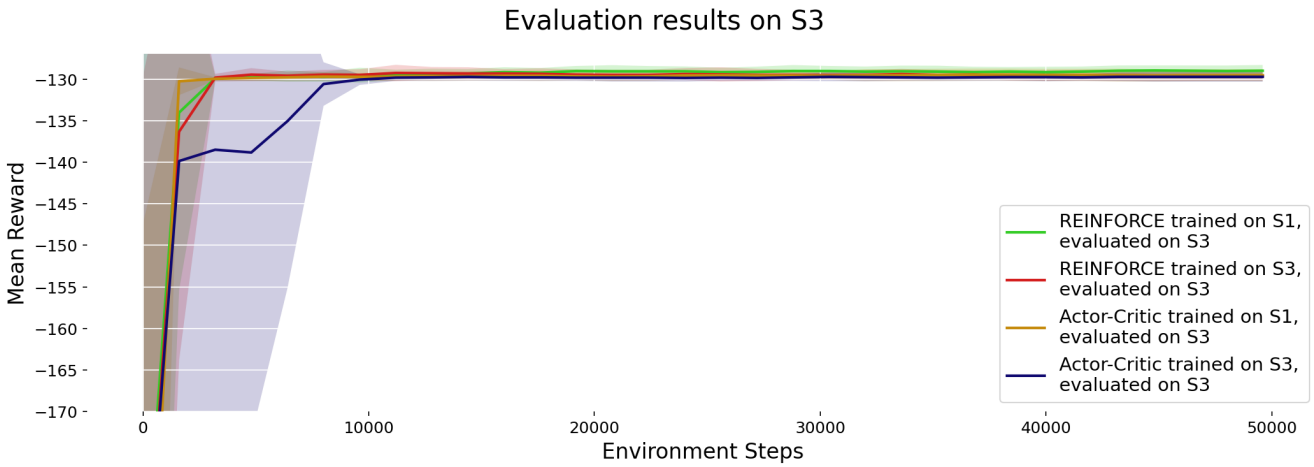
6.3 Alternative baseline design

Here we report the learning curves comparing two training variants of the ACR-Policy: REINFORCE with a greedy baseline and an Actor-Critic variants of the attention models. We evaluated the two methods under different training settings to analyze their performance under both in-scenario and cross-scenario conditions.

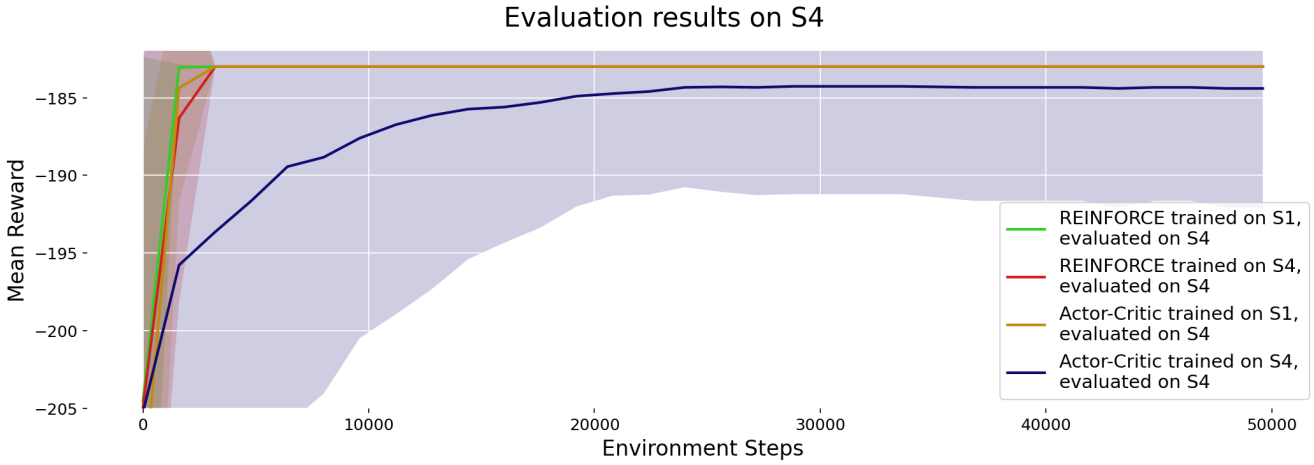
Single-crane variant For evaluation on Scenario S3, both algorithms are trained on S1 and S3, respectively. For evaluation on Scenario S4, both algorithms are trained on S1 and S4.

As can be observed in Figures 2a and 2b, the proposed REINFORCE with a greedy baseline exhibits lower variance compared to the Actor-Critic algorithm on most scenarios.

All four algorithms on evaluation scenario S3 are able to find the optimal solution, as suggested by almost zero-variance from step 8000 onward, proving that this environment instance is relatively easy to solve by the proposed attention-based policy network. In this scenario, REINFORCE with a greedy baseline trained on S1 performs the best, showing no decrease in final performance in later training phases and also exhibiting the lowest variance. It is closely followed by Actor-Critic trained on S1, which learns the fastest at early time steps, but has a small drop off in performance at later time steps, and REINFORCE with a greedy baseline trained on S3, which performs very similarly, in terms of convergence speed and variance. The worst performing variant is Actor-Critic trained on S3, showing the slowest convergence and the largest variance, with high deviations lasting on average until time step 8000.



(a) Training on scenarios S1 and S3, evaluated on scenario S3.



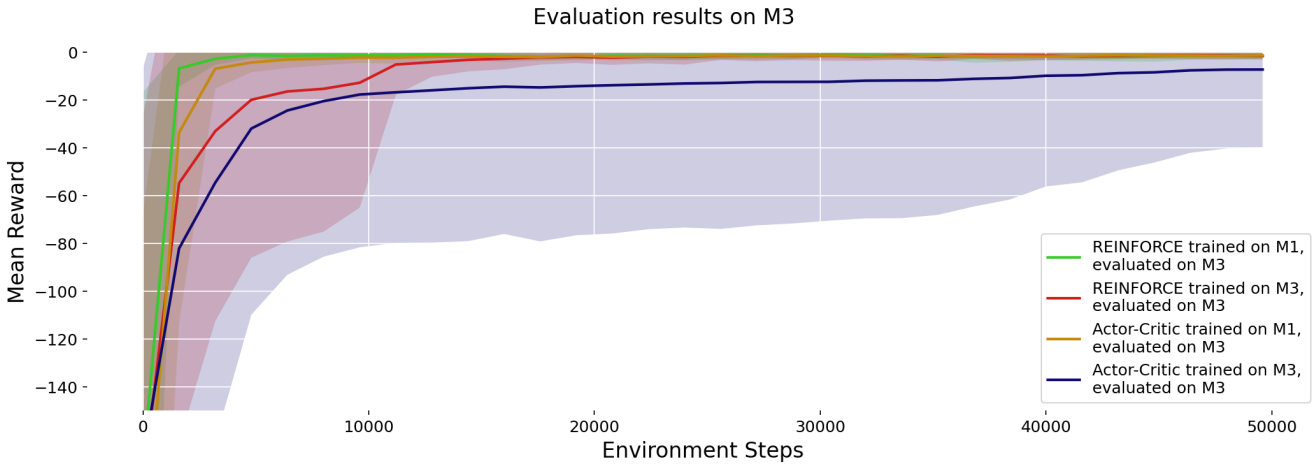
(b) Training on scenarios S1 and S4, evaluated on scenario S4.

Figure 2: Comparison of the REINFORCE with greedy baseline and Actor-Critic algorithms under varying training and evaluation scenarios for the single-crane variant of the environment. In scenario S3 the best performing variant is Actor-Critic trained on S1, while in scenario S4, REINFORCE trained on S1 seems to perform better. Actor-Critic trained on S4 shows notably higher variance than other algorithms.

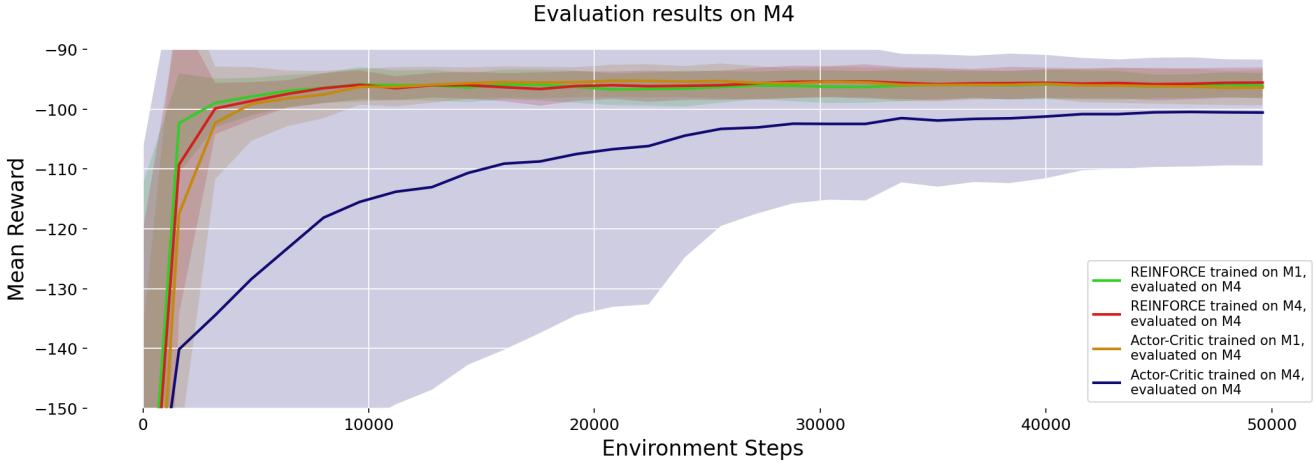
Considering evaluation scenario S4 we observe that it is notably more difficult for the agents due to the increased number of container groups, which necessitates more frequent reshuffles and better long-term planning in order to minimize the cost. The algorithm that shows the fastest learning and lowest variance is REINFORCE with a greedy baseline trained on S1, followed by Actor-Critic trained on S1, with marginally slower convergence speed but notably higher variance. Notably, despite the larger training environment for REINFORCE with a greedy baseline trained on S4, it shows competitive performance to the two aforementioned algorithms, with marginally higher variance and slightly slower convergence speed. Finally, the algorithm that performs the worst on this scenario is the Actor-Critic trained on S4, showing a large variance in the entire range of environment time steps and significantly slower learning speed.

Actor-Critic's higher variance could result from the amplification of the value and policy

network's mismatch in selecting the best action and evaluating it accurately on larger training scenarios. Considering a case where an agent is trained on environments with 45 containers, it receives more frequent feedback as opposed to environments with 200 containers, due to the terminal reward structure in the environment.



(a) Training on scenarios M1 and M3, evaluated on scenario M3.



(b) Training on scenarios M1 and M4, evaluated on scenario M4.

Figure 3: Comparison of the REINFORCE with greedy baseline and Actor-Critic algorithms under varying training and evaluation scenarios for the multi-crane variant of the environment. The best performing agent on both evaluation scenarios is REINFORCE trained on M1. REINFORCE trained on M3 shows notably higher variance in scenario M3, compared to other experiments comparing REINFORCE variants trained on larger problem sizes.

Multi-crane variant The multi crane comparison was performed as follows: for the comparison on M3 each agent was trained on scenarios M1 and M3, respectively, while for the comparison on M4 both agents were trained on M1 and M4.

Figure 3a shows the performance of the ACR-Policy variants: REINFORCE with greedy baseline and Actor-Critic on evaluation scenario M3. REINFORCE trained on M1 exhibits the quickest convergence speed and lowest variance, followed by Actor-Critic trained on the same environment,

which takes notably longer to converge to the final reward of around 0, as around timestep 100000 it keeps learning, while REINFORCE has already converged. The two remaining algorithms show a notable difference in performance, with REINFORCE trained on M3 displaying much higher variance and lower mean reward even around timestep 10000, compared to the two aforementioned algorithms that almost converge at this timestep. The learning speed of this algorithm is also notably lower, however the algorithm can finally converge to similar solutions as the variants trained on M1. Finally, Actor-Critic trained on M3 exhibits the least promising performance, showing high variance throughout the entire time step window, and never fully converging to similar mean reward values as the other variants.

Figure 3b depicts the performance of the agents during evaluation on scenario M4. Similarly to 3a the best performing agent is REINFORCE trained on M1, however, it is very closely followed by REINFORCE trained on M3 unlike in the previous setting. Both agents seem to show similar variance, while the variant trained on M1 converges slightly faster. The high variance problem of REINFORCE trained on M3 does not seem to be present in this experimental setting, however even the best performing agent, which is REINFORCE trained on M1 shows noticeably higher variance, compared to evaluation scenario M3. The third best-performing agent is Actor-Critic trained on M1, showing similarly slower convergence speed in this evaluation scenario. It does however converge to an equally good final mean reward as the two REINFORCE variants. The Actor-Critic variant trained on M4 exhibits high variance across all time steps, failing to converge to a competitive final mean reward like other algorithms tested in this setting.

REINFORCE performs very well overall on the medium sized multi-crane scenarios, showing fast convergence and robust, low-variance solutions, apart from the variant trained and evaluated on scenario M3. Overall, it performs better than Actor-Critic, and the greedy baseline is shown to be an effective choice at reducing variance in these problem settings.

Actor-Critic variants, even when trained on small scenarios tend to converge slower than the corresponding REINFORCE variant with greedy baseline. This could be due to the fact that the value network needs more time to stabilize state value-function estimates, compared to a greedy baseline that effectively compares the algorithm's performance with itself, albeit from before, which is similar to findings from the single crane experiment.

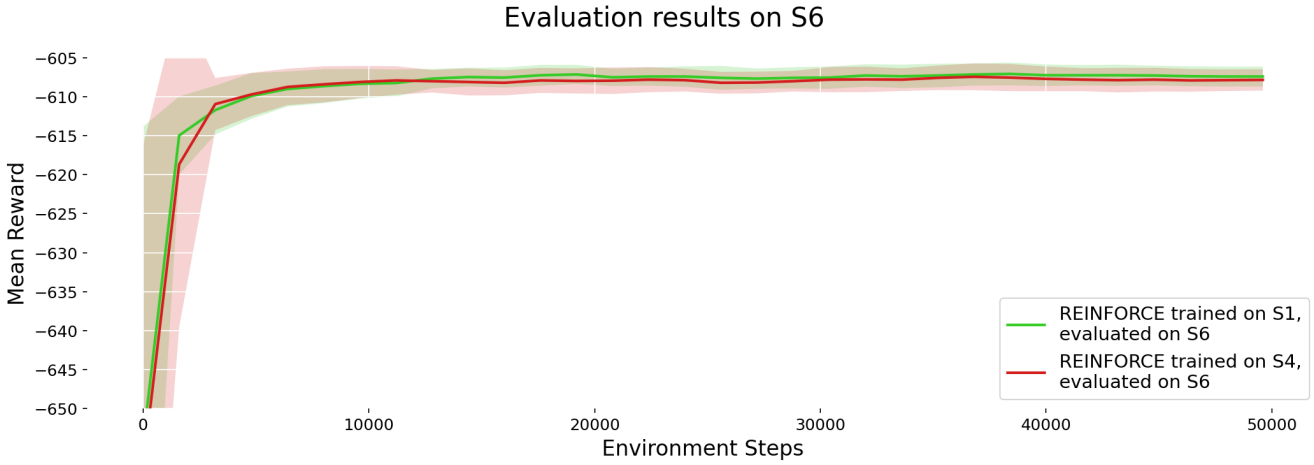
6.4 Generalization capability

In order to test performance on larger environment sizes, we inspect the performance of the best performing algorithm on scenarios S5, S6, as well as M5 and M6. It is important to note that the agents were not trained on these scenarios, due to higher computational costs and higher variance, they only serve as further points of evaluation to test generalization.

Single-crane variant The proposed ACR-Policy was trained on scenarios S1 and S4 using the better-performing greedy baseline. Figures 4a and 4b show the results of the agent evaluations on scenarios S5 and S6, respectively. Both of these scenarios are challenging to solve, due to sparser rewards and larger problem instances, forcing the agents to balance intermediate gain from reshuffling containers at each step and the long term impact on the arrangement of containers in the yard.

In scenario S5, the agents perform similarly, ACR-Policy trained on S1 shows faster learning and lower variance, but has a small drop in performance in later time steps, REINFORCE trained on

(a) Evaluation on scenario S5.



(b) Evaluation on scenario S6.

Figure 4: Comparison of the REINFORCE with greedy baseline trained on scenarios S1 and S4 when evaluating on scenarios S5 and S6. REINFORCE trained on S4 outperforms REINFORCE trained on S1 in evaluation scenario S5. The opposite is true when evaluating on scenario S6.

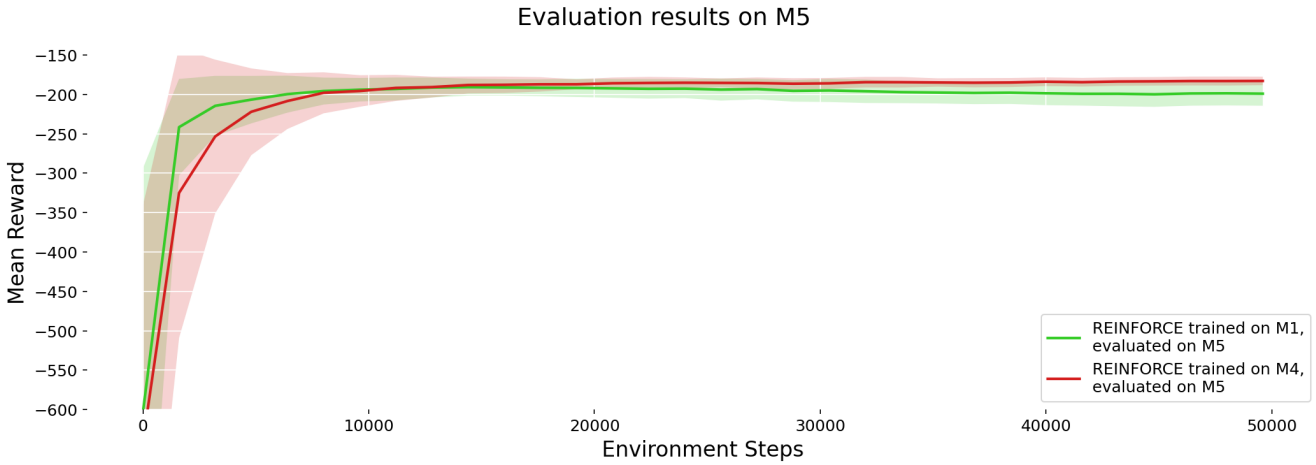
S4, on the contrary, exhibits slower learning and higher variance but is able to find better solutions at the end of its training. The difference with ACR-Policy trained on S4 is similar to Figure 2b, as this algorithm again shows slightly slower convergence and higher variance.

However, considering scenario S6, ACR-Policy trained on S1 outperforms the variant trained on S4, in terms of learning speed, while variance of both algorithms remains comparable. What is interesting to note is that the agents after converging to their best possible reshuffle counts both show higher variations in the later time steps compared to evaluating on S5. This could be attributed to the relatively high number of container groups, which naturally requires more reshuffles and can lead to more difficult layouts, as the choice of a container and its group at each step is more important than with fewer groups.

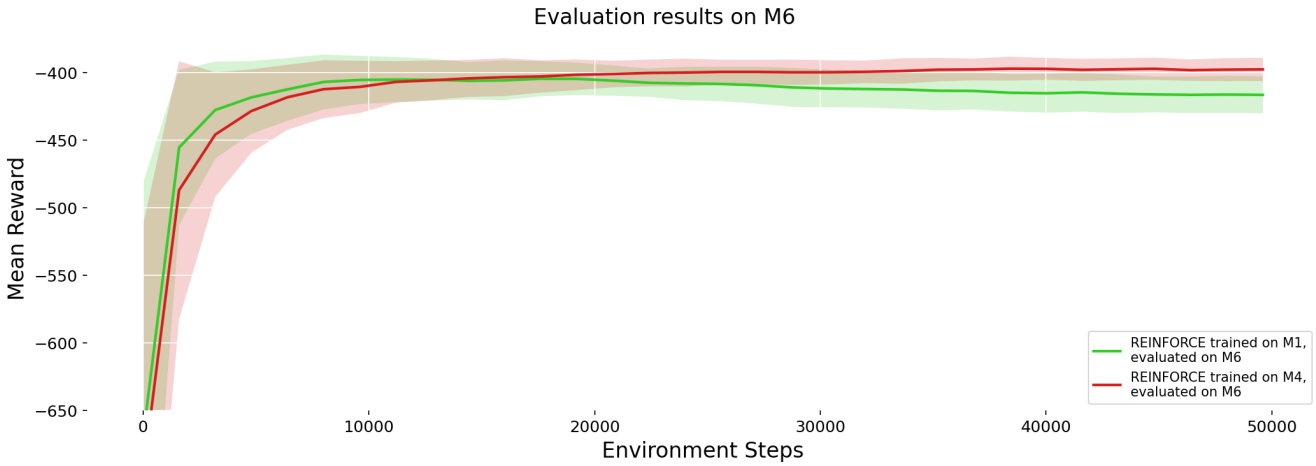
These results show that even during the evaluation of challenging problem instances our architecture is able to capture the essential problem information from smaller training scenarios,

successfully generalizing to larger yard and vessel sizes, higher group numbers, and more containers to stow. The ACR-Policy demonstrates promising zero-shot performance on different scenarios.

When it comes to computation cost and performance trade-off, we recommend training the architecture on as small problem instances as possible, as the final performance is then usually the most promising or at least very competitive with models trained on larger instances, which require more video memory.



(a) Evaluation on scenario M5.



(b) Evaluation on scenario M6.

Figure 5: Comparison of the REINFORCE with greedy baseline trained on scenarios M1 and M4 when evaluating on scenarios M5 and M6. Unlike in the single-crane environment, REINFORCE trained on M1 shows higher variance and decrease in performance on later time steps, compared to REINFORCE trained on M4, which shows stronger performance in both M5 and M6.

Multi-crane variant The REINFORCE algorithm with the greedy baseline was trained on scenario M1 and M4 and evaluated on scenarios M5 and M6.

Figure 5a shows the performance on evaluation scenario M5. REINFORCE trained on M1 learns faster initially and shows lower variance compared to the agent trained on M4, however, around

time step 20000 it starts to gradually decrease in performance, reaching a noticeably lower reward that the agent trained on a larger scenario. The agent trained on M4 shows moderate variance at time steps below 15000, however it converges to a promising solution in the later time steps. These results are different from these shown in Figure 4, where the agent trained on a smaller scenario performs better.

Considering scenario M6 we can inspect Figure 5b, where the agent trained on M1 again shows faster initial learning. However, it seems to have comparable variance to the variant trained on M4 in early time steps, while after time step 20000 it shows an even worse reduction in performance than in Figure 5a, as well as a higher amount of variance. REINFORCE trained on M4 does not suffer from the same shortcomings as the agent trained on M1, converging to a stable final mean reward, without a decrease at later time steps.

An important observation from the multi-crane experiments is the reduction in final performance of the variant trained on M1. This could be attributed to the agent overfitting on the relatively small problem instance, which naturally negatively impacts its ability to generalize. Another problem is the inherent lack of diversity in the data collected on the small problem instance, where especially the degrees of freedom present in the different number of groups of containers are much lower for M1 with 3 groups, compared to a large scenario such as M6 with 16 groups. To a lesser extent this problem could also be a consequence of insufficient hyperparameter tuning, as we made the assumption that scenario M4 is representative for all multi-crane experiments, however, it is possible that the learning rate might be too high for scenario M1, which potentially causes the agent to oscillate around the optimum, as shown by a reduction in performance and higher variance.

Comparing the results to the single-crane experiments we can conclude that the agents trained on very small scenarios do not always show good generalization beyond a certain problem size, as it was shown in scenarios M5 and M6. Instead we would opt for a representative scenario that is shown to generalize well to larger problems. However, if the problem is sufficiently simple to solve, like the single-crane variant, it is recommended to train the agent on a smaller environment if possible to reduce computational cost.

7 Conclusion

We have compared the ACR-Policy variants to Deep-RL and a GA implementation under different training scenarios and a novel environment formulation allowing for flexible container selection. It was shown that ACR-Policy finds more competitive solutions than previous baselines on medium size environments (S3, S4, M3, M4), while offering substantial performance improvements in comparison to a single-thread Genetic Algorithm implementation. ACR-Policy with greedy baseline has demonstrated promising zero-shot performance, when training on a different scenario than the one used for evaluation.

In smaller senarios (S1, S2, M1, M2) we found that the baselines usually offered stronger performance, but they quickly became obsolete as the problem size increased, suggesting a capacity-complexity trade-off. The transformer model's capacity is likely too large for small training instances, where it can overfit to spurious correlations in the specific training instances, while in larger, much more difficult, training instances the ability to attend over all slots simultaneously allows the architecture get a notable performance improvement.

The architecture was then compared with a value network baseline, which was shown to perform

worse than the greedy baseline used in REINFORCE. The algorithms took longer to converge and were prone to extremely high variance for larger training scenarios, showing that the greedy baseline can be successful in reducing variance and improving training stability, while the value network can be prone to instability during training, taking longer to converge to high-quality solutions.

When attempting to test the performance on larger scenarios with 400 containers, which are more challenging to solve due to sparser rewards, requiring the agents to balance intermediate gains from reshuffling and long term planning of the yard layout, it was found that the performance on small and medium scenarios does not always translate to robust performance on much larger, unseen instances, where usually an agent trained on a medium scenario such as S4 and especially M4, outperformed the agent trained on a small environment.

In order to reduce variance on these large scenarios it is vital to carefully select a representative scenario that can generalize to the desired setting. Considering the available degrees of freedom in the data such as container groups has an important impact on final agent performance, as even if the features are normalized, the agent sees fewer different values and might find it difficult to extrapolate the finer details, which are not present during training. On the single-crane variant, however, using a small training scenario such as S1 still offers competitive performance with a non-negligible improvement in training speed as it requires fewer hardware resources.

8 Future work

The baseline experiments showed a potential lack of model capacity for PPO and DQN, when trained on larger scenarios such as S4 and M4. A possible future research direction could include adapting the implementations of these algorithms to employ a similar attention-based architecture [13].

Our environment is also designed as a very simplified version of a vessel and yard simulator. The reshuffling operation is abstracted by simply shifting the blocking containers down by one tier, without considering relocations to other stacks. [9] formulates two possible reshuffling strategies, namely the nearest and lowest stack strategy, where the blocking container is moved to the closest stack in the vincinity of the current stack, without blocking the current container, and a lowest stack strategy where the blocking container is moved to the stack(s) in the yard that currently have the fewest containers. Adapting these strategies to the current environment formulation could give more accurate reshuffle estimates and allow for comparison with previous work.

Similarly, another limitation with our environment formulation is in the case of the multi-crane variant. The current environment reward function formulation does not enforce efficient, parallel crane utilization, as the objective is the same as in the single crane variant: minimization of the total reshuffling cost. As noted by [21], reducing total reshuffles does not enforce maximum crane utilization, however, formulating a reward function that accounts for crane parallelization usually also leads to a reduction in reshuffle counts.

The agent evaluation procedure simply samples a single evaluation scenario instance and saves the agent performance at each evaluation step. Considering multiple samples or a search algorithm used for inference could give better solutions without a higher training budget [4][13].

A final consideration is to redesign the decoder to allow for fully auto-regressive generation of entire solutions, which will enable the algorithm to use an even more powerful baseline, used by [14], where the agent is compared against an average of different decoding trajectories of itself.

This baseline was shown to reduce variance and improve performance beyond the greedy baseline used by [13]. However an ongoing challenge with this design consideration is employing action masking when autoregressively selecting the next container.

References

[1] Daniela Ambrosino, Anna Sciomachen, and Elena Tanfani. Stowing a containership: The master bay plan problem. *Transportation Research Part A: Policy and Practice*, 38:81–99, 02 2004.

[2] Mordecai Avriel and Michal Penn. Exact and approximate solutions of the container ship stowage problem. *Computers Industrial Engineering*, 25(1):271–274, 1993.

[3] Anibal Azevedo, Luiz Leduino, Antonio Chaves, and A.C. Moretti. Solving the 3d stowage planning problem integrated with the quay crane scheduling problem by representation by rules and genetic algorithm. *Applied Soft Computing*, 65, 04 2018.

[4] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning, 2017.

[5] JaeHyeok Cho and NamKug Ku. Developing a container ship loading-planning program using reinforcement learning. *Journal of Marine Science and Engineering*, 12(10), 2024.

[6] M. G. Garratt. Ro-ro versus lo-lo: a matter of scale. *Maritime Policy & Management*, 7(4):223–232, 1980.

[7] Junliang He, Leijie Zhang, Yiyun Deng, Hang Yu, Mingzhong Huang, and Caimao Tan. An allocation approach for external truck tasks appointment in automated container terminal. *Adv. Eng. Inform.*, 55(C), January 2023.

[8] Yunqi Huang, Nishith Chennakeshava, Alexis Carras, Vladislav Neverov, Wei Liu, Aske Plaat, and Yingjie Fan. A benchmark study of deep reinforcement learning algorithms for the container stowage planning problem. In *Dynamics in Logistics (LDIC 2026)*, Lecture Notes in Logistics, pages 103–118. Springer, 2026.

[9] Mingjun Ji, Wenwen Guo, Huiling Zhu, and Yongzhi Yang. Optimization of loading sequence and rehandling strategy for multi-quay crane operations in container terminals. *Transportation Research Part E: Logistics and Transportation Review*, 80:1–19, 2015.

[10] Bo Jin and Shunji Tanaka. An exact algorithm for the unrestricted container relocation problem with new lower bounds and dominance rules. *European Journal of Operational Research*, 304(2):494–514, 2023.

[11] Bo Jin, Wenbin Zhu, and Andrew Lim. Solving the container relocation problem by an improved greedy look-ahead heuristic. *European Journal of Operational Research*, 240:837–847, 02 2015.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[13] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems!, 2019.

[14] Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning, 2021.

[15] Jaejin Lee, Seho Kee, Mani Janakiram, and George Runger. Attention-based reinforcement learning for combinatorial optimization: Application to job shop scheduling problem, 2024.

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[17] Dario Pacino, Alberto Delgado, Rune Jensen, and Tom Bebbington. Fast generation of near-optimal plans for eco-efficient stowage of large container vessels. volume 6971, pages 286–301, 01 2011.

[18] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[20] Yifan Shen, Ning Zhao, Mengjue Xia, and Xueqiang Du. A deep q-learning network for ship stowage planning problem. *Polish Maritime Research*, 24, 11 2017.

[21] Woo-Jin Shin, Inguk Choi, Sang-Hyun Cho, and Hyun-Jung Kim. Learning to retrieve containers: A scale-diverse deep reinforcement learning approach for the container retrieval problem. *Transportation Research Part C: Emerging Technologies*, 183:105496, 2026.

[22] Dirk Steenken, Stefan Voß, and Robert Stahlbock. Container terminal operation and operations research - a classification and literature review. *OR Spectrum*, 26(1):3–49, 2004.

[23] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 2 edition, 2018.

[24] S. Theofanis, M. Boile, and M. M. Golias. Container terminal berth planning: Critical review of research approaches and practical challenges. *Transportation Research Record*, 2100(1):22–28, 2009.

[25] Kevin Tierney, Dario Pacino, and Rune Møller Jensen. On the complexity of container stowage planning problems. *Discrete Applied Mathematics*, 169:225–230, 2014.

[26] UNCTAD. Review of maritime transport 2018. *UN Trade & Development*, 2018.

[27] UNCTAD. Review of maritime transport 2024. *UN Trade & Development*, 2024.

[28] Jaike van Twiller, Djordje Grbic, and Rune Møller Jensen. Towards a deep reinforcement learning model of master bay stowage planning. In Joachim R. Daduna, Gernot Liedtke, Xiaoning Shi, and Stefan Voß, editors, *Computational Logistics*, pages 105–121, Cham, 2023. Springer Nature Switzerland.

[29] Jaike van Twiller, Agnieszka Sivertsen, Dario Pacino, and Rune Møller Jensen. Literature survey on the container stowage planning problem. *European Journal of Operational Research*, 317(3):841–857, 2024.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[31] Ning Wang, Xinyu Ma, Ruobing Yang, and Jingjing Hu. Optimization of partially ordered container relocation problem with rolled containers. *Computers Operations Research*, 191:107434, 2026.

[32] Peixiang Wang, Qihang Xu, Yufei Li, Qunlong Chen, Jinghan Tao, Wei Qin, Heng Huang, and Ying Zou. Learning-based hybrid algorithms for container relocation problem with storage plan. *Transportation Research Part E: Logistics and Transportation Review*, 197:104048, 2025.

[33] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[34] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992.

[35] I D Wilson and P A Roach. Container stowage planning: a methodology for generating computerised solutions. *Journal of the Operational Research Society*, 51(11):1248–1255, 2000.

[36] Ning Zhao, Yuechao Guo, Tianyu Xiang, Mengjue Xia, Yifan Shen, and Chao Mi. Container Ship Stowage Based on Monte Carlo Tree Search. *Journal of Coastal Research*, 83(sp1):540 – 547, 2019.

9 Appendix

Hyperparameters tuned for each agent for the single crane experiment are shown in Table 4, while the multi-crane hyperparameters are shown in Table 5.

Optimal Hyperparameter	Agent Type			
	DQN	PPO	REINFORCE	Actor-Critic
Learning rate	0.000109	0.000271	0.000276	0.000557
Gamma (γ)	0.9561	0.9159	0.9280	0.9129
Entropy coefficient	—	0.000120	0.0409	0.00302
Baseline update p-value	—	—	0.3171	—
GAE λ	—	0.8622	—	—
Clip coefficient	—	0.3989	—	—
Value function coefficient	—	0.9584	—	—
Max gradient norm	—	8.4768	—	—
Number of minibatches	—	4	—	—
Update epochs	—	5	—	—
Soft update (τ)	0.0982	—	—	—
Target network frequency	1400	—	—	—
Batch size	256	—	—	—
Exploration fraction	0.7444	—	—	—
Final exploration rate	0.0216	—	—	—
Train frequency	4	—	—	—

Table 4: Optimal hyperparameter values for the single crane experiments for DQN, PPO, REINFORCE and Actor-Critic. The agents were tuned on S4.

Optimal Hyperparameter	Agent Type			
	DQN	PPO	REINFORCE	Actor-Critic
Learning rate	0.000688	0.000820	0.000434	0.000425
Gamma (γ)	0.9470	0.9636	0.9614	0.9462
Entropy coefficient	—	0.00278	0.00425	0.01279
Baseline update p-value	—	—	0.1277	—
GAE λ	—	0.9428	—	—
Clip coefficient	—	0.2914	—	—
Value function coefficient	—	0.5173	—	—
Max gradient norm	—	0.9655	—	—
Number of minibatches	—	16	—	—
Update epochs	—	6	—	—
Soft update (τ)	0.5127	—	—	—
Target network frequency	1500	—	—	—
Batch size	128	—	—	—
Exploration fraction	0.8321	—	—	—
Final exploration rate	0.0296	—	—	—
Train frequency	4	—	—	—

Table 5: Optimal hyperparameter values for the multi-crane experiments for DQN, PPO, REINFORCE and Actor-Critic. The agents were tuned on M4.