Solving the Pod Repositioning Problem with Deep Reinforced Adaptive Large Neighborhood Search

Lin Xie[0000−0002−3168−4922], Hanyi Li

Brandenburg University of Technology Cottbus-Senftenberg, Cottbus, D-03046, Germany
`lin.xie@b-tu.de`

Abstract. The Pod Repositioning Problem (PRP) in Robotic Mobile Fulfillment Systems (RMFS) involves selecting optimal storage locations for pods returning from pick stations. This work presents an improved solution method that integrates Adaptive Large Neighborhood Search (ALNS) with Deep Reinforcement Learning (DRL). A DRL agent dynamically selects destroy and repair operators and adjusts key parameters such as destruction degree and acceptance thresholds during the search. Specialized heuristics for both operators are designed to reflect PRP-specific characteristics, including pod usage frequency and movement costs. Computational results show that this DRL-guided ALNS outperforms traditional approaches such as cheapest-place, fixed-place, binary integer programming, and static heuristics. The method demonstrates strong solution quality and illustrating the benefit of learning-driven control within combinatorial optimization for warehouse systems.

Keywords: Pod repositioning problem · Deep reinforcement learning · Adaptive large neightborhood search · Warehouse logistics

1 Introduction

Robotic Mobile Fulfillment Systems (RMFS) have revolutionized warehouse operations by enabling mobile robots to transport storage pods (also known as racks) between storage areas and operational stations such as picking or replenishment stations. A critical operational decision in RMFS is determining the optimal storage location for a pod after it has been processed at a station. This decision, referred to as the *Pod Repositioning Problem (PRP)* or *Rack Storage Assignment Problem*, significantly impacts robot travel distances, inventory accessibility, and overall system throughput. For example, pods that are not expected to be needed soon should be placed farther from stations to preserve nearby storage locations for high-frequency or urgent items.

The PRP has been formally defined in a deterministic context by Krenzler et al. [2], who proposed a sequential decision-making model aimed at minimizing cumulative travel costs associated with pod storage assignments. Their work evaluated various strategies, including binary integer programming, cheapest-place heuristics, fixed-place policies, and a tetris-like heuristic. Similarly, Weidinger et al. [10] addressed the similar storage assignment problem, formulating it as an interval scheduling problem and employing mixed-integer programming to optimize pod movements and workload balancing.

Further contributions to deterministic storage assignment strategies include Merschformann et al. [5] considered both active and passive repositioning scenarios. Yuan et al. [11] developed a velocity-based storage assignment policy for semi-automated storage systems, demonstrating the benefits of class-based storage over random assignment. Zhuang et al. [12] investigated the pod storage and automated guided vehicle (AGV) task assignment problem, proposing a meta-heuristic decomposition approach to optimize pod retrieval and repositioning.

While these deterministic approaches have proven effective in structured environments, they often rely on static heuristics or require extensive parameter tuning. To enhance adaptability and performance, meta-heuristic frameworks like Adaptive Large Neighborhood Search (ALNS) [8] offer flexible solution methods by iteratively destroying and repairing parts of a solution to explore the search space effectively. Reijnen et al. [6] extended this concept by integrating Deep Reinforcement Learning (DRL) into the ALNS framework, allowing dynamic selection and configuration of heuristics during the search process. Although this DRL-guided ALNS (DR-ALNS) has shown promise in routing problems, its application to storage assignment in RMFS remains unexplored.

In parallel, DRL has been applied to dynamic and stochastic variants of warehouse storage problems. Rimélé et al. [7] trained a deep Q-network to learn dynamic storage policies in simulated RMFS environments, while Teck et al. [9] proposed a DRL-based method for real-time inventory pod storage and replenishment under uncertainty. These studies emphasize reactive decision-making in stochastic settings. DRL has shown promising results in warehouse operations, including order batching [1] and picker routing [3], [4].

Despite the advancements in dynamic and stochastic approaches, many warehouse operations still function under deterministic or semi-deterministic conditions, where order streams and system states are predictable over short horizons. Leveraging deterministic models allows for exploiting known demand patterns and system structures, facilitating more efficient and reliable storage assignment strategies. Building upon the deterministic framework established by Krenzler et al. [2], this study aims to enhance storage assignment performance through adaptive metaheuristic methods without introducing stochastic complexities.

Research Gap and Contribution. To the best of our knowledge, the integration of DRL into metaheuristic frameworks like ALNS has not been applied to the deterministic PRP in RMFS. Existing DRL approaches focus on dynamic, real-time decision-making, while traditional deterministic methods rely on static heuristics.

This paper makes the following contributions:

- We adapt the DR-ALNS framework to the deterministic pod repositioning problem, demonstrating its applicability and effectiveness in structured warehouse environments.
- We develop and formalize domain-specific destroy and repair heuristics tailored to the PRP, enhancing the flexibility and performance of the ALNS framework.
- Through computational experiments, we compare our approach against established baselines, including cheapest-place, fixed-place, and binary integer programming methods, showcasing improved solution quality and adaptability.

This work complements the existing literature by demonstrating that adaptive metaheuristic search, guided by reinforcement learning, can yield substantial improvements even within deterministic, rule-driven storage environments.

2   Problem Description: Deterministic Pod Repositioning Problem

We follow the deterministic formulation of the Pod Repositioning Problem (PRP) proposed by Krenzler et al. [2], where the goal is to assign returning pods to storage locations over time to minimize total robot travel costs. Each placement decision affects future availability and retrieval efficiency due to the shared and time-constrained nature of storage locations.

Let T denote the planning horizon and S the set of storage locations. At each time step $t \in \{0, \ldots, T-1\}$, a set of pods $\mathcal{P}_t$ returns to storage. For each pod $p \in \mathcal{P}_t$, we assign a storage location $a_t(p) \in S$. Let $d(p)$ be the next departure time of p, and define the travel costs:

- $c_{\text{store}}(p, s, t)$: cost to store pod p at location s at time t,
- $c_{\text{retrieve}}(p, s, d(p))$: cost to retrieve pod p from location s at time $d(p)$.

Objective Function

The objective is to minimize the total storage and retrieval costs across all time steps:

$$\min \sum_{t=0}^{T-1} \sum_{p \in \mathcal{P}_t} \left[c_{\text{store}}(p, a_t(p), t) + c_{\text{retrieve}}(p, a_t(p), d(p)) \right]$$

Constraints

1. **No overlap:** A storage location cannot be used by multiple pods during overlapping occupancy periods:

$$\forall s \in S, \quad \text{at most one } p \text{ such that } a_t(p) = s \text{ and } t < d(p)$$

2. **Feasibility:** Each pod must be assigned to an available location at its return time:

$$a_t(p) \in S \quad \text{and } s = a_t(p) \text{ must be unoccupied for } [t, d(p))$$

Heuristic Consideration

In practice, it is beneficial to prioritize frequently used pods for locations that are closer to pick stations to reduce average retrieval costs. Let $f(p)$ denote the access frequency of pod p and $S_{\text{near}} \subseteq S$ denote the set of preferred (i.e., near-access) storage locations. A heuristic strategy may enforce:

$$\text{if } f(p) \text{ is high, then prefer } a_t(p) \in S_{\text{near}}$$

This prioritization is often embedded in the design of repair heuristics (e.g., ABC-based placement).

Discussion

The PRP is a dynamic, time-indexed assignment problem where the cost and feasibility of each decision are interdependent. Although the system evolves deterministically, its constrained nature creates long-range dependencies between placement decisions. This motivates the use of metaheuristics such as ALNS, which can iteratively refine global placement patterns, and learning-based extensions like DRL, which can adaptively control heuristic behavior in context.

3 Adaptive Large Neighborhood Search for the Pod Repositioning Problem

This section details our implementation of ALNS, including the destroy and repair heuristics, solution feasibility and acceptance criteria.

3.1 ALNS Framework

ALNS iteratively refines an initial solution through a cycle of destruction and repair operations. A set of heuristics for each phase is maintained, with adaptive weight updates promoting those that consistently lead to improved solutions. The general process is shown in Algorithm 1.

Algorithm 1 Adaptive Large Neighborhood Search

1: **Input:** Initial solution s_0, temperature schedule $(T_{start}, T_{stop}, \alpha)$, Markov chain length M, acceptance probability p_{accept}, heuristic weights w_d, w_r
2: Set $s \leftarrow s_0$, $s^* \leftarrow s_0$
3: **for** $i = 1$ to M **do**
4:     Select destroy heuristic $d \in \mathcal{D}$ with probability proportional to w_d
5:     $s' \leftarrow d(s)$
6:     Select repair heuristic $r \in \mathcal{R}$ with probability proportional to w_r
7:     $s'' \leftarrow r(s')$
8:     **if** $\text{cost}(s'') < \text{cost}(s^*)$ **then**
9:         $s^* \leftarrow s''$
10:        Accept $s \leftarrow s''$ unconditionally
11:        Update w_d, w_r (global improvement)
12:    **else if** $\text{cost}(s'') < \text{cost}(s)$ or $\text{rand}() < p_{accept}$ **then**
13:        Accept $s \leftarrow s''$
14:        Update w_d, w_r (local or accepted worse solution)
15:    **else**
16:        Update w_d, w_r (no improvement)
17:    **end if**
18:    Update temperature $T \leftarrow \max(T \cdot \alpha, T_{stop})$
19: **end for**
20: **return** s^*

To guide the selection of destroy and repair heuristics, the ALNS framework maintains a set of weights w_d and w_r associated with each heuristic. At the beginning of each iteration, heuristics are chosen probabilistically according to these weights. After the resulting solution is evaluated, the weights are updated using a score-based reward scheme inspired by the original ALNS method [8].

Let $\sigma \in \{0, 1, 2, 3\}$ be the reward assigned based on the outcome of an iteration:

- $\sigma = 3$ if the new solution improves the global best (global improvement),
- $\sigma = 2$ if it improves the current solution (local improvement),
- $\sigma = 1$ if it is accepted but not better (accepted worse),
- $\sigma = 0$ if it is rejected.

The weight of a selected heuristic h is then updated as:

$$w_h \leftarrow \lambda \cdot w_h + (1 - \lambda) \cdot \sigma$$

where $\lambda \in [0, 1]$ is a reaction factor controlling how strongly new feedback influences the weight. This adaptive mechanism enables the algorithm to favor heuristics that consistently lead to better solutions, while still allowing underperforming heuristics occasional influence for exploration.

The performance of ALNS critically depends on the design of effective *destroy* and *repair* heuristics. In this section, we expand on the heuristics implemented in our approach to the PRP, including their rationale, working principles, and pseudocode.

3.2   Destroy Heuristics

Destroy heuristics remove a subset of the pod-to-location assignments from the current solution s over a sequence of iterations. This creates space for exploring alternative placements and escaping local optima. Let $I_{\text{destroy}} \subseteq \{0, \ldots, N-1\}$ be the set of iterations selected for destruction.

Random Destruction This heuristic selects a consecutive range of $k = \lfloor \text{DoD} \cdot N \rfloor$ iterations uniformly at random and removes the pod assignments in those iterations. It encourages exploration by applying changes across arbitrary regions of the solution. We use a parameterized Degree of Destruction (DoD)[1] to control the intensity of each destroy operation.

Algorithm 2 Random Destruction

1: **function** DESTROYRANDOM(s, DoD)
2:      $k \leftarrow \lfloor \text{DoD} \cdot N \rfloor$
3:      Choose $i_{\text{start}} \in \{0, \ldots, N-k\}$ uniformly
4:      $I_{\text{destroy}} \leftarrow \{i_{\text{start}}, \ldots, i_{\text{start}} + k - 1\}$
5:      **for** $i \in I_{\text{destroy}}$ **do**
6:          Remove pod assignment in iteration i from s
7:      **end for**
8:      **return** s'
9: **end function**

High-Cost Destruction This heuristic targets the most expensive region of the solution. It identifies a window of k consecutive iterations that cumulatively contribute the highest cost, and removes their assignments. It supports intensification by attacking problematic areas.

---

[1] The Degree of Destruction (DoD) is a scalar in $(0, 1]$ that determines how many iterations are selected for removal in the destroy phase. If N is the total number of iterations, then $\lfloor \text{DoD} \cdot N \rfloor$ entries are destroyed. Higher DoD values increase exploration, while lower values favor local refinement.

Algorithm 3 High-Cost Destruction

1: **function** DESTROYHIGHCOST(s, DoD)
2:     $k \leftarrow \lfloor \text{DoD} \cdot N \rfloor$
3:     **for** $i = 0$ to $N - k$ **do**
4:         $C_i \leftarrow \sum_{j=0}^{k-1} \text{cost}(s[i+j])$
5:     **end for**
6:     $i^* \leftarrow \arg\max_i C_i$
7:     $I_{\text{destroy}} \leftarrow \{i^*, \ldots, i^* + k - 1\}$
8:     **for** $i \in I_{\text{destroy}}$ **do**
9:         Remove pod assignment in iteration i
10:     **end for**
11:     **return** s'
12: **end function**

3.3 Repair Heuristics

After destruction, repair heuristics reconstruct the partial solution s' by assigning storage placements to the unassigned pods at destroyed iterations I_{destroy}. All heuristics rely on the `FeasibleLocations` function from Section 3.4 to ensure valid placements.

Tetris-Inspired Repair This repair heuristic is loosely inspired by the Tetris algorithm from Krenzler et al. [2], which places returning pods greedily in time order by selecting the lowest-cost feasible location at each step. In contrast, our approach introduces a two-phase structure to improve placement quality.

In Phase 1, we target iterations with the highest placement costs—those likely to create inefficiencies if handled later—and assign them greedily to feasible locations. This anticipates future congestion and mimics the idea of placing the "bulkiest" pieces first, analogous to challenging blocks in Tetris. In Phase 2, the remaining iterations are sorted by pod frequency and urgency (departure time), and are assigned to cost-effective locations using the same greedy placement rule.

Algorithm 4 Tetris-Inspired Repair

1: **function** REPAIRTETRIS(s', I_{destroy})
2:     $U \leftarrow I_{\text{destroy}}$
3:     Sort U by descending placement cost
4:     **for** $i \in U$ **do**
5:         $P_i \leftarrow$ `FeasibleLocations`(i)
6:         **if** $P_i \neq \emptyset$ **then**
7:             Assign pod in i to highest-cost location in P_i
8:             Remove i from U
9:         **end if**
10:     **end for**
11:     Sort U by pod frequency and next departure time
12:     **for** $i \in U$ **do**
13:         $P_i \leftarrow$ `FeasibleLocations`(i)
14:         **if** $P_i \neq \emptyset$ **then**
15:             Assign pod in i to lowest-cost location in P_i
16:         **end if**
17:     **end for**
18:     **return** s''
19: **end function**

ABC Priority Repair Pods are categorized into three classes based on their usage frequency: Class A (high frequency, accounting for 70% of total usage), Class B (medium frequency, 20%), and Class C (low

frequency, 10%). Class A pods are assigned to the most optimal storage locations, Class B to moderately favorable ones, and Class C to the least optimal, reflecting their relative operational importance.

Algorithm 5 ABC Priority Repair

1: **function** REPAIRABC(s', I_{destroy}, frequency)
2:     **for** $i \in I_{\text{destroy}}$ **do**
3:         $P_i \leftarrow$ FeasibleLocations(i)
4:         Determine class of pod p (A/B/C)
5:         **if** Class A **then**
6:             Choose lowest-cost location
7:         **else if** Class B **then**
8:             Choose second-best location
9:         **else**
10:             Choose third-best location
11:         **end if**
12:         Assign pod p to selected location
13:     **end for**
14:     **return** s''
15: **end function**

Lowest-Cost Repair This greedy heuristic always selects the feasible placement with the lowest cost for each destroyed iteration, independent of pod characteristics.

Algorithm 6 Lowest-Cost Repair

1: **function** REPAIRLOWESTCOST(s', I_{destroy})
2:     **for** $i \in I_{\text{destroy}}$ **do**
3:         $P_i \leftarrow$ FeasibleLocations(i)
4:         **if** $P_i \neq \emptyset$ **then**
5:             Assign pod in i to lowest-cost location
6:         **end if**
7:     **end for**
8:     **return** s''
9: **end function**

Random Repair To maintain solution diversity, this heuristic randomly selects a feasible placement for each destroyed iteration. It is especially helpful for escaping local optima.

Algorithm 7 Random Repair

1: **function** REPAIRRANDOM(s', I_{destroy})
2:     **for** $i \in I_{\text{destroy}}$ **do**
3:         $P_i \leftarrow$ FeasibleLocations(i)
4:         **if** $P_i \neq \emptyset$ **then**
5:             Randomly assign pod to a location in P_i
6:         **end if**
7:     **end for**
8:     **return** s''
9: **end function**

3.4   Feasibility Checks

Feasibility during repair is evaluated by three interdependent functions:

- **RecallPlaces:** This function simulates the warehouse state at a given iteration and tracks pod locations. It's especially important for evaluating feasible placements in the future and finding previous positions of a target pod.

- **IsPlaceFeasible:** This function checks if a place is feasible for storing a pod, i.e., it's unoccupied during the relevant time interval.

- **FeasibleLocations:** This function evaluates all storage locations to identify those that are feasible for a pod returning at a certain iteration. It uses RecallPlaces to simulate warehouse state and IsPlaceFeasible to validate placements. It also estimates cost (including transit cost if applicable).

Algorithm 8 RecallPlaces

1: **function** RECALLPLACES(solution, iteration, arrivals, departures, target_pod)
2:     Initialize pod placements and station queues
3:     prev_location $\leftarrow$ None
4:     **for** $x = 0$ to iteration **do**
5:         Retrieve pod arriving and departing at x
6:         Update warehouse configuration: remove departing pod, add arriving pod
7:         **if** target_pod is the one departing at x **then**
8:             prev_location $\leftarrow$ recorded location before departure
9:         **end if**
10:        Update Station1 and Station2 based on queue dynamics
11:    **end for**
12:    **return** current warehouse configuration at iteration, prev_location, Station1, Station2
13: **end function**

Algorithm 9 IsPlaceFeasible

1: **function** ISPLACEFEASIBLE(config, solution, iteration, place_id, next_departure, destroy_set, departures)
2:     **if** place is currently occupied in config **then**
3:         **return** False
4:     **end if**
5:     **if** next departure exists **then**
6:         **for** each future iteration j before next_departure $- 1$ **do**
7:             **if** another pod is assigned to place_id at j and $j \notin$ destroy_set **then**
8:                 **return** False
9:             **end if**
10:        **end for**
11:    **else**
12:        **for** future iterations j **do**
13:            **if** another pod is assigned to place_id and $j \notin$ destroy_set **then**
14:                **return** False
15:            **end if**
16:        **end for**
17:    **end if**
18:    **return** True
19: **end function**

Algorithm 10 FeasibleLocations

1: **function** FEASIBLELOCATIONS(iteration, solution, I_{destroy}, arrivals, departures, warehouse)
2:     Get returning pod p and origin station s_{from} from departure at iteration
3:     Get next departure of p as t_{next}
4:     (config, prev_loc, _) ← RecallPlaces(solution, iteration, arrivals, departures, p)
5:     Initialize empty list feasible_placements ← []
6:     **for** each storage place q in warehouse **do**
7:         **if** IsPlaceFeasible(config, solution, iteration, q, t_{next}, I_{destroy}, departures) **then**
8:             Compute base cost: $c \leftarrow c^{\text{from}}(s_{\text{from}}, q)$
9:             **if** t_{next} exists **then**
10:                 $c \leftarrow c + c^{\text{to}}(q, s_{\text{next}})$
11:             **end if**
12:             Append (q, c) to feasible_placements
13:         **end if**
14:     **end for**
15:     **return** feasible_placements and prev_loc
16: **end function**

3.5   Acceptance Criteria

We adopt Simulated Annealing (SA) as the acceptance criterion. A new solution s'' is always accepted if it improves the current best. Otherwise, it is accepted probabilistically:

$$P(\text{accept}) = \begin{cases} 1 & \text{if } \text{cost}(s'') < \text{cost}(s) \\ \exp\left(-\frac{\Delta}{T}\right) & \text{otherwise} \end{cases}$$

where $\Delta = \text{cost}(s'') - \text{cost}(s)$ and T is the current temperature. This strategy ensures a balance between exploration and exploitation throughout the search.

4   Deep Reinforcement Learning for ALNS

Traditional ALNS relies on weight-based selection of destroy and repair heuristics, where weights are updated based on historical performance (as shown in Section 3). While effective, this method does not explicitly incorporate the current search context or problem dynamics. To enhance the performance of ALNS in complex and dynamic settings, we incorporate a Deep Reinforcement Learning (DRL) agent (introduced by Reijnen et al.[6]) to control the adaptive selection of operators and parameters during the search process. This hybrid approach, referred to as DR-ALNS, leverages learning-based decision-making to guide destroy and repair operations as well as acceptance parameters in an online manner. DR-ALNS models the ALNS configuration process as a Markov Decision Process (MDP), allowing a DRL agent to make context-aware decisions at each iteration.

Our implementation follows the general DR-ALNS framework proposed by Reijnen et al. [6], adapted specifically to the pod repositioning context by integrating domain-specific components such as warehouse simulation and feasibility-aware placement evaluation.

4.1   MDP Formulation

Our implementation casts ALNS for pod repositioning as a Markov Decision Process with the following elements:

– **State** s_t: a fixed-length vector comprising
    1. Normalized temperature T/T_{start}.
    2. Previous cost change $\Delta_{t-1}/C^{\text{best}}$.
    3. Current gap to best $(C^{\text{curr}} - C^{\text{best}})/C^{\text{best}}$.
    4. Normalized destroy-operator weights (one entry per destroy heuristic).

    5. Normalized repair-operator weights (one entry per repair heuristic).
    6. Current cost ratio $C^{\mathrm{curr}}/C^{\mathrm{best}}$.
    7. Absolute best cost C^{best}.
    8. Cost gap.
    9. Progress fraction $t/T_{\max}$.
Together, these components inform the agent about how "hot" the annealer is, how near the current solution is to the global best, which heuristics have shown promise, and how much of the search budget remains.

- **Action** a_t: a single integer encoding a triple $(d_t,\ r_t,\ \delta_t)$, where
  - $d_t \in \{\text{destroy heuristics}\}$,
  - $r_t \in \{\text{repair heuristics}\}$,
  - $\delta_t \in \{\text{DoD levels}\}$ "DoD" stands for "Degree of Destruction". In the context of ALNS, it specifies how much of the current solution is removed during the destroy phase.

  Decoding into (d_t, r_t, δ_t) identifies which destroy operator to apply, which repair operator to follow, and how many pods to remove in that iteration.

- **Reward** r_t: computed immediately after each destroy–repair iteration combined with simulated-annealing acceptance. It consists of:
  1. Normalized cost improvement $\left(\Delta_t/C^{\mathrm{init}}\right)$.
  2. New-best bonus $+1.0$ if the accepted solution establishes a new global best.
  3. Fluctuation penalty -0.5 if the last three cost deltas follow "down–up–down."
  4. Repair-failure penalty -0.2 if the repair step was infeasible.
  5. Rejection penalty -0.1 if simulated annealing rejects the candidate.
  6. Exploration bonus $+0.1 \times (T/T_{\mathrm{start}})$ if a non-improving but feasible solution is accepted.

  This composite reward encourages steady cost reduction, discourages zig-zag behavior, penalizes infeasibility and rejections, and lightly rewards early exploration.

- **Transition** $(s_t, a_t) \to (s_{t+1}, r_t)$:
  1. Decode: the integer a_t into a chosen destroy operator d_t, repair operator r_t, and DoD level δ_t.
  2. Destroy phase: remove δ_t pods from the current solution using d_t. If no pods can be removed, immediately assign $r_t = -1.0$ and keep $s_{t+1} = s_t$.
  3. Repair phase: rebuild a full solution with r_t, yielding cost difference $\Delta_t = C^{\mathrm{old}} - C^{\mathrm{new}}$ and a failure indicator.
  4. Simulated-annealing acceptance: accept unconditionally if $\Delta_t > 0$; otherwise accept with probability $\exp\left(\Delta_t/T\right)$. If accepted and $C^{\mathrm{new}} < C^{\mathrm{best}}$, update the global best.
  5. Operator-weight update: assign a score $\sigma \in \{0, 1, 2, 3\}$ to both d_t and r_t (0 for failure/rejection, 1 if accepted without improvement, 2 if accepted with improvement but not new-best, 3 if new-best). Then update each selected weight via

  $$w \ \leftarrow \ 0.95\,w \ + \ 0.05\,\sigma, \quad w \geq 0.1.$$

  Record Δ_t in a length-3 buffer for possible fluctuation penalty.
  6. Reward computation: compute r_t as defined above.
  7. Temperature cooling: $T \leftarrow \max(T_{\mathrm{stop}}, T \times \text{decrease\_factor})$.
  8. Increment iteration: $t \leftarrow t + 1$.
  9. Termination check: stop if $T \leq T_{\mathrm{stop}}$ or $t \geq T_{\max}$.
  10. Next observation: s_{t+1} is constructed using the same nine-element structure.

- **Policy** π_θ: a parameterized mapping (e.g., via Proximal Policy Optimization) from each observed s_t to a distribution over the discrete actions $\{0, \ldots, |\mathcal{D}| \times |\mathcal{R}| \times |\Delta| - 1\}$. Over training episodes, π_θ learns which destroy/repair/DoD combinations yield the largest cumulative reward (i.e., the lowest repositioning cost).

4.2 DRL Integration in ALNS for the Pod Repositioning Problem

Building on the MDP definition, DRL integration proceeds as follows:

1. **Composite Action Selection.** At each iteration, the agent samples a_t from $\pi_\theta(\cdot \mid s_t)$, simultaneously choosing a destroy operator, a repair operator, and a DoD level. This replaces the traditional weight-based roulette selection with a learned policy that directly accounts for the current search context.

2. **Online Parameter Adaptation.** By observing normalized operator weights in each s_t, the policy can bias toward heuristics that have performed well recently, while retaining the capacity to explore under-utilized options thanks to the enforced lower bound on weights. The annealing temperature T and the iteration index t further inform the policy about exploration–exploitation trade-offs.

3. **Reward-Driven Learning.** The shaped reward balances immediate cost improvements, sparse global-best discoveries, penalties for unproductive patterns, and exploration bonuses. This encourages the agent to discover long-term strategies (e.g., applying a heavy destruction early, switching heuristics mid-search, or gradually tightening acceptance criteria) rather than solely focusing on short-term gains.

4. **Simulated-Annealing Mechanism.** The MDP's transition incorporates an SA acceptance rule: if $\Delta_t > 0$, accept; otherwise accept with probability $\exp(\Delta_t/T)$. This stochastic element preserves exploration capabilities even when the policy proposes non-improving moves, especially at higher temperatures. Over time, as T decays, the agent's decisions become more greedy.

5. **Training Loop.** An off-the-shelf DRL algorithm (e.g., Proximal Policy Optimization) interacts with the MDP:
   - Observe s_t.
   - Sample $a_t \sim \pi_\theta(\cdot \mid s_t)$.
   - Execute the combined destroy–repair–SA step, receive r_t and s_{t+1}.
   - Store the transition $(s_t, a_t, r_t, s_{t+1}, \text{done})$.
   - Periodically update θ to maximize expected discounted return.

   Over many episodes—each consisting of up to $T_{\max}$ destroy–repair iterations—the policy learns which operator sequences and DoD choices yield the best repositioning cost reductions in varied search contexts.

By combining composite action selection, operator-weight feedback, SA acceptance, and tailored reward shaping, this framework realizes a lightweight, efficient DR-ALNS approach for the Pod Repositioning Problem.

5   Computational Results

This section first describes the test instances, then outlines the parameter learning process for both ALNS and DR_ALNS, and concludes with a computational comparison against all baseline methods.

5.1   Instance Description

To evaluate the performance of our proposed ALNS algorithm, we adopt two benchmark instances introduced by Krenzler et al. [2], which model pod repositioning within an RMFS. These instances are well-suited for assessing both the effectiveness and scalability of pod placement strategies under realistic warehouse conditions.

The *small instance* comprises a storage area with 10 locations and 10 pods, operating over 1,000 discrete time steps. The warehouse layout is one-dimensional and symmetric with respect to two identical picking stations. This configuration allows for clear visualization of pod dynamics and cost patterns, making it ideal for qualitative algorithm analysis and debugging.

The *medium instance* represents a significantly larger and more complex environment, featuring 504 storage locations, 441 pods, and two asymmetric picking stations. The system is simulated over 20,000 time steps, during which approximately 20,000 repositioning decisions are made—excluding a short initialization period required to fill the station queues. Pod departures follow a weighted stochastic process that favors high-frequency pods and prioritizes one of the stations with higher throughput, thereby emulating non-uniform demand patterns observed in real-world e-commerce warehouses.

Both instances are based on a deterministic model in which pod movements incur cost according to Manhattan distances between storage locations and pick stations. The problem setting strictly adheres to passive repositioning: a pod is only moved after serving a pick station, with no predictive or anticipatory relocation. This setup enables direct comparison with the baseline heuristics proposed in Krenzler et al. [2], including *Cheapest Place*, *Fixed Place*, *Fixed Place (Approximate)*, *Random Place*, *Binary Integer Programming*, *BIP-Iterative*, *Genetic Algorithms*, and the *Tetris heuristic*. *Cheapest Place* selects the nearest available location; *Fixed Place* assigns each pod a dedicated storage slot; *Fixed Place (Approximate)* uses usage frequency to

assign pods to ranked zones; *Random Place* chooses locations randomly; *Binary Integer Programming* (BIP) offers an exact, optimization-based solution over the full time horizon; *BIP-Iterative* solves smaller subproblems sequentially to handle scalability; *Genetic Algorithms* use evolutionary search to improve placements; and the *Tetris heuristic* prioritizes frequently used pods for optimal positions based on occupation intervals.

5.2   Parameter Learning for ALNS and DR_ALNS

All experiments of ALNS and DR_ALNS were conducted on a workstation equipped with an Intel Core i9-10850K CPU @ 3.60 GHz, 32 GB RAM, and an NVIDIA RTX 3060 GPU.

ALNS: Simulation-Based Parameter Optimization

To ensure a fair comparison with the DR_ALNS, we employ a simulation-based procedure to calibrate the parameters of the ALNS and generate a high-quality reference solution derived exclusively from the ALNS framework. The tuning process focuses on four main parameters of the algorithm:

- $T_{\mathbf{start}}$: Initial temperature for the simulated annealing acceptance criterion.
- $T_{\mathbf{stop}}$: Final temperature that determines when cooling ends.
- **MLength**: Markov chain length, i.e., the number of iterations at a fixed temperature.
- **DecreaseFactor**: Cooling rate α that controls how quickly T_{start} is reduced.

The parameter ranges were selected based on preliminary tests and expert judgment:

$$T_{\mathrm{start}} \in \{5, 7.5, 10, 12.5, 15\},$$
$$T_{\mathrm{stop}} \in \{0.1, 0.3, 0.5, 0.7, 0.9\},$$
$$\mathrm{MLength} \in \{30, 40, 50\},$$
$$\mathrm{DecreaseFactor} \in \{0.76, 0.80, 0.84, 0.88, 0.92, 0.95\}.$$

Fifty random parameter combinations were tested in the small instance. Each was evaluated by running ALNS and recording the total cost and runtime. To encourage exploration, the acceptance criterion included a 25% chance of accepting worse solutions. Additionally, when no improvement occurred for several consecutive iterations, a reinitialization of parameters was triggered to escape potential local optima.

The best configuration found was:

$$T_{\mathrm{start}} = 12.5, \quad T_{\mathrm{stop}} = 0.1, \quad \mathrm{MLength} = 30, \quad \mathrm{DecreaseFactor} = 0.95$$

This configuration was used in the final experimental evaluation due to its superior balance between solution quality and runtime.

DR_ALNS: Policy Training

Policy learning was conducted exclusively on the small instance. A total of 20,000 training timesteps were used, with each episode comprising 1,000 ALNS iterations. Training was performed in Python 3.11.4 with Stable Baselines3 (v2.5.0), and required approximately 1.1 hours to complete. The resulting PPO agent was saved as a model checkpoint for subsequent evaluation and inference. The main PPO hyperparameters were as follows: a learning rate of 1×10^{-3}, batch size of 128, $n_{\mathrm{steps}} = 2{,}048$, and an entropy coefficient of 0.01 to encourage exploration.

For inference, we evaluate the trained DR-ALNS PPO agent in the same warehouse environment configuration used during training. The inference environment employs 1,000 ALNS iterations, an initial temperature T_{start} of 1.0, a stopping temperature T_{stop} of 0.001, and a temperature decrease factor of 0.98.

5.3   Comparison with baselines

Figures 1 and 2 present the total costs of various heuristics as a percentage relative to the random placement baseline (100%), evaluated on small and medium warehouse instances, respectively.

On the **small instance** (Figure 1), *BIP*, which computes the optimal solution, is the top performer on the **small instance** at **61.85%**, closely followed by *DR_ALNS* (62.38%) and *ALNS* (63.68%). Classic heuristics such as *Cheapest Place* and *Tetris* remain competitive, while *Fixed Place* incurs significantly higher costs (93.48%), indicating limited adaptability in smaller-scale settings.

On the **medium instance** (Figure 2), *DR_ALNS* achieves the best result with a cost of **59.9%**, followed by *ALNS* (65.24%) and *Tetris* (73.8%). Traditional heuristics such as *Cheapest Place* (91.01%) and *Fixed Place* variants (74.75%) show noticeably higher costs.

Overall, *DR_ALNS* consistently ranks among the best-performing methods across both instance sizes, highlighting the benefits of learning-based and online adaptive control. In contrast, traditional heuristics exhibit reduced effectiveness, particularly as problem complexity increases.

In terms of efficiency, all baseline methods—except for *BIP* (on the medium instance) and the *Genetic Algorithm*—complete within one minute for both instance sizes. *ALNS* solves the small instance in 103 seconds and the medium instance in about 27 minutes. The inference times of *DR_ALNS* are 74 and 400 seconds for the small and medium instances, respectively.

Notably, both *ALNS* and *DR_ALNS* maintain strong performance on the medium instance using parameters tuned only on the small instance (via simulation optimization for ALNS and policy training for DR_ALNS). This demonstrates good generalization capability and robustness—especially for *DR_ALNS*, which adapts effectively to larger problem scales without the need for re-tuning.
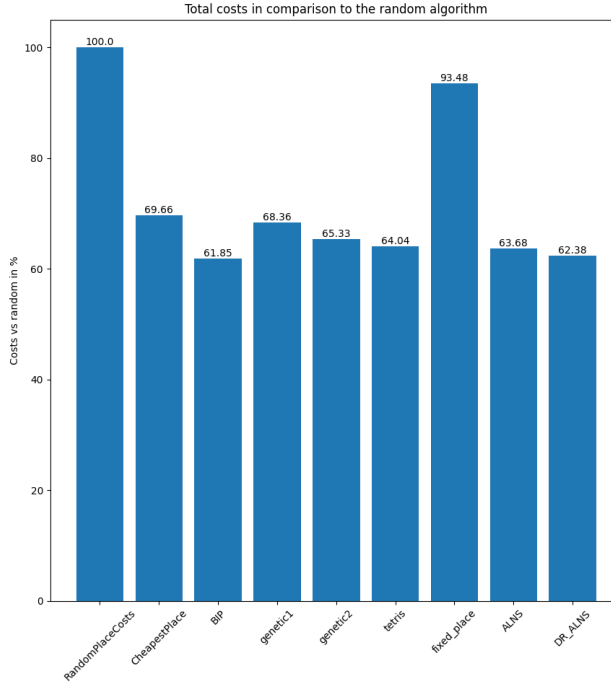

Fig. 1. Comparison of ALNS, DR_ALNS with baseline methods for the small instance.

6   Conclusion and Outlook

In this work, we adapted the DR-ALNS framework to address the deterministic Pod Repositioning Problem (PRP), demonstrating its effectiveness in structured warehouse environments. By incorporating domain-
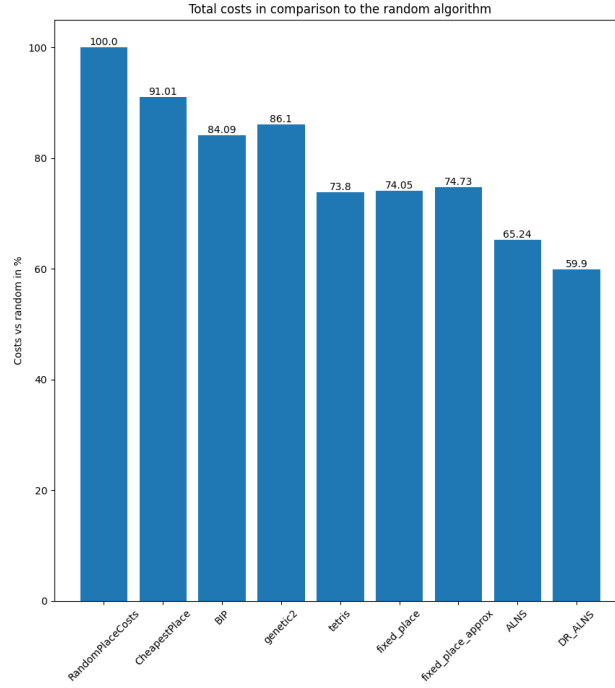
Fig. 2. Comparison of ALNS, DR_ALNS with baseline methods for the medium instance.

specific destroy and repair heuristics, we enhanced the flexibility of the ALNS approach to better match the operational characteristics of the PRP. Computational experiments across different instance sizes show that DR-ALNS consistently delivers comparable or superior performance to classical baselines such as *Cheapest Place*, *Fixed Place*, *Tetris*, and *Binary Integer Programming*, achieving both high solution quality and robustness. Notably, the policy used in DR-ALNS was trained solely on the small instance and successfully generalized to larger instances without re-training, highlighting the adaptability and transferability of the learned policy across problem scales.

As a next step, future work could explore extending the approach to stochastic or real-time settings, or integrating it with other warehouse decision-making layers such as order picking or inventory management, to further enhance its practical applicability.

References

1. Cals, B., Zhang, Y., Dijkman, R., van Dorst, C.: Solving the online batching problem using deep reinforcement learning. Computers & Industrial Engineering **156**, 107221 (2021). https://doi.org/10.1016/j.cie.2021.107221
2. Krenzler, R., Xie, L., Li, H.: Deterministic pod repositioning problem in robotic mobile fulfillment systems. arXiv preprint arXiv:1810.05514 (2018), https://arxiv.org/abs/1810.05514
3. Luttmann, L., Xie, L.: Neural combinatorial optimization on heterogeneous graphs: An application to the picker routing problem in mixed-shelves warehouses. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS). vol. 34, pp. 351–359 (2024). https://doi.org/10.1609/icaps.v34i1.31494
4. Luttmann, L., Xie, L.: Learning to solve the min-max mixed-shelves picker-routing problem via hierarchical and parallel decoding (2025), https://arxiv.org/abs/2502.10233

5. Merschformann, M., Boysen, N., Briskorn, D.: Active repositioning of storage units in robotic mobile fulfillment systems. Transportation Research Part E: Logistics and Transportation Review **111**, 1–16 (2018). https://doi.org/10.1016/j.tre.2018.03.011

6. Reijnen, R., Zhang, Y., Lau, H.C., Bukhsh, Z.: Online control of adaptive large neighborhood search using deep reinforcement learning. Artificial Intelligence **319** (2024)

7. Rimélé, J., Courtois, A., Risso, J.: E-commerce warehousing: Learning a storage policy. European Journal of Operational Research **291**(1), 123–134 (2021). https://doi.org/10.1016/j.ejor.2020.08.038

8. Ropke, S., Pisinger, D.: An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation Science **40**(4), 455–472 (2006). https://doi.org/10.1287/trsc.1050.0135

9. Teck, K., Li, Y., Lin, Z.: Deep reinforcement learning for the real-time inventory rack storage assignment and replenishment problem. To appear in European Journal of Operational Research (2025)

10. Weidinger, F., Boysen, N., Briskorn, D.: Storage assignment with rack-moving mobile robots in kiva warehouses. Transportation Science **52**(5), 1479–1495 (2018). https://doi.org/10.1287/trsc.2017.0766

11. Yuan, R., Graves, S.C., Cezik, T.: Velocity-based storage assignment in semi-automated storage systems. Production and Operations Management **28**(2), 354–373 (2019). https://doi.org/10.1111/poms.12937

12. Zhuang, Y., Zhou, Y., Hassini, E., Yuan, Y., Hu, X.: Rack retrieval and repositioning optimization problem in robotic mobile fulfillment systems. Transportation Research Part E: Logistics and Transportation Review **167**, 102920 (2022). https://doi.org/10.1016/j.tre.2022.102920