University of Bergen

Department of informatics
&
Geophysical Institute

Hyperheuristic Frameworks for Combinatorial Optimization Problems using Deep Reinforcement Learning

Master's Thesis

Author: Akilavan Sivakumaran

Supervisor: Ahmad Hemmati

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2024

Abstract

Many metaheuristic frameworks exist for solving different combinatorial optimization problems. Despite formulating general strategies that can be applied to many problems, they often rely on problem-specific implementation. Hyperheuristic frameworks attempt to fully generalize the solution method by only relying on general search information for decision making. The addition of Deep Reinforcement Learning (DRL) in a hyperheuristic framework provides the opportunity of learning complex relations between different actions and their effect relative to the state of the search. When it is used for selection of heuristics, it is important to mitigate the opportunity for reward hacking by carefully designing the reward function to be as representative of our objective as possible. This thesis proposes two hyperheuristic frameworks using DRL, with a new reward function for heuristic selection that is based on the percentage improvement compared to the initial solution. Deep Reinforcement Learning Hyperheuristic Plus (DRLH$^+$) combines this DRL heuristic selection with the acceptance strategy of simulated annealing. Dual-Network Deep Reinforcement Learning Hyperheuristic (D^{2}RLH) combines the DRL heuristic selection with a second DRL agent for acceptance. The frameworks are tested by solving instances of the Pickup and Delivery Problem with Time Windows, and consistently perform well on large problem sizes. The reward function is shown to improve upon the reward function of Deep Reinforcement Learning Hyperheuristic (DRLH) by making gradual and consistent improvements throughout the search, and is able to adjust the strategy to account for extended searches.

Acknowledgements

This master's thesis marks the end of a five year integrated master's program in Energy. I am grateful to everyone who have been part of my experiences as a student. I would like to thank my supervisor, Ahmad Hemmati, for his support and invaluable advice throughout the last year.

I would also like to thank my extended family and friends who I am extremely lucky to have in my life. I would like to mention my close friends Simen, Kamilla and Håkon for their support and company throughout my five years as a student. I am grateful for having worked on this thesis surrounded by Terese, Sander, Ingrid, Mari and Emilie, as their presence has made every day enjoyable. Finally, I express my gratitude to all fellow students who I consider treasured friends.

This work is part of the Ocean Charger project, mainly funded by the Norwegian Research Council (Project No. 340936).

Akilavan Sivakumaran

13 June, 2024

List of Acronyms and Abbreviations

ALNS	Adaptive Large Neighborhood Search.
CVRP	Capacitated Vehicle Routing Problem.
D^{2}RLH	**D**ual-Network **D**eep **R**einforcement **L**earning **H**yperheuristic.
DRL	Deep Reinforcement Learning.
DRLH	Deep Reinforcement Learning Hyperheuristic.
DRLH$^+$	**D**eep **R**einforcement **L**earning **H**yperheuristic Plus.
DRLMA	Deep Reinforcement Learning Move Acceptance.
GAE	Generalized Advantage Estimate.
PDPTW	Pickup and Delivery Problem with Time Windows.
PJSP	Parallel Job Scheduling Problem.
PPO	Proximal Policy Optimization.
TRPO	Trust Region Policy Optimization.
URS	Uniform Random Selection.
VRP	Vehicle Routing Problem.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The emerging consequences of climate change have emphasized the need for development of sustainable energy sources. The member states of the United Nations share a plan of action for sustainable development, stating 17 sustainable development goals [21]. Several of the sustainability goals tie into reduction of emission and the development of sustainable energy and infrastructure.

To work towards reducing emissions, it is important for any process that consumes energy to utilize resources efficiently. Even the process of producing clean energy itself can benefit from optimization. As an example, the operation of offshore wind farms requires maritime infrastructure. The Ocean Charger project aims to develop low-emission operation with battery-driven vessels, by connecting the electric vessels to the power grid of the wind farm [22]. To keep resource usage as low as possible, it is essential to optimize the routes of the vessels during operation. Formulating this type of scenario as a general combinatorial optimization problem allows us to develop methods that are applicable to any field where a similar problem arises.

The Pickup and Delivery Problem with Time Windows (PDPTW) is a routing problem that can represent a wide range of cases where some resource needs to be transported. It is a generalized problem that can handle a large number of locations and resources, and take

both distances and time windows into account. The objective of the PDPTW is to minimize the cost, mainly a result of traveled distance, of the transportation of resources by a fleet of available vessels. Each load has to be picked up at one location and delivered at a second location. Extensive research has been conducted on PDPTW, as the general formulation with few assumptions makes it widely applicable.

In this thesis, we aim to further develop solution methods for PDPTW. Rather than creating problem-specific solutions, the goal is to solve the problem with a generalized method that should also be suitable for combinatorial optimization problems in general.

1.2 Approach

For many combinatorial optimization problems, exact methods become excessively time-consuming for large-scale instances. A metaheuristic is a problem-specific implementation of guidelines provided by a problem-independent framework and can be applied to conduct an efficient and practical search for solutions by following some strategy. The purpose of a metaheuristic is to achieve reasonable solutions in a short time frame. A prominent example of a metaheuristic known to perform well on combinatorial optimization problems is Adaptive Large Neighborhood Search (ALNS) [13]. This algorithm repeatedly selects from a set of heuristics that can be applied to the current solution, and continuously adjusts the probability of selecting each heuristic according to the quality of the solutions they produce.

Hyperheuristics are another class of solution methods, which try to achieve a higher level of abstraction by avoiding problem-specific implementations. These search methods attempt to generalize to a problem-independent level by only relying on the progress of the search in decision making, and not dynamics that are specific to the problem. Many hyperheuristics provide a strategy for selecting heuristics during the search, instead of directly searching the solution space. By applying hyperheuristic frameworks to solving the PDPTW we can contribute to both the specific problem at hand, and methods for combinatorial optimiziation in general.

A development that has shown great potential in recent years is applying deep reinforcement learning to hyperheuristic frameworks. Deep reinforcement learning is a subfield of machine learning that attempts to maximize a reward signal by incorporating deep neural

networks into decision-making. By allowing an intelligent agent to make decisions solely based on the state of the search, it can learn complex strategies that can adapt according to the circumstances. Kallestad [10] used this concept to develop the hyperheuristic framework Deep Reinforcement Learning Hyperheuristic (DRLH), which replaces the adaptive heuristic selection of ALNS with a deep neural network. The framework was shown to outperform ALNS on all four optimization problems that were tested.

Isaksen [9] continued the use of deep reinforcement learning for hyperheuristics by proposing Deep Reinforcement Learning Move Acceptance (DRLMA), a framework that uses an intelligent agent to decide if found solutions should be accepted or not. Every time a heuristic is used to produce a new solution, the acceptance agent decides if the new solution should replace the current solution. The possibility of accepting worse solutions plays an important role in escaping local optima, and training a network to handle the acceptance can lead to more complex behavior. DRLMA was shown to work well together with DRLH on two different optimization problems, replacing the previously used acceptance strategy from the metaheuristic simulated annealing.

To realize the full potential of reinforcement learning, it is important to ensure that the rewards that we maximize are in line with the real objective we want to achieve. If this aspect is not carefully considered, the agent might learn behavior that maximizes the rewards in an unexpected way that does not align with our objective. The reward function used in DRLH awards points if a new solution is found that is better than the previous solution, and a larger amount of points if it is better than the best solution found so far. As improvements are rewarded regardless of the extent, the agent might learn to make multiple small improvements that allow it to receive the rewards several times. Even though this would encourage the agent to improve the solution, the agent would not benefit from improving it efficiently. A heuristic search aims to find good solutions efficiently, and it would be a major advantage to encourage the agent to make larger improvements.

In this thesis, the design of the reward function is a hyperheuristic component that is especially emphasized. We want to assess a new reward function for selecting heuristics that is designed to be as closely related to the objective as possible, hopefully mitigating the opportunity for unexpected behavior.

This leads to the introduction of a new hyperheuristic framework using deep reinforcement learning, namely the **D**eep **R**einforcement **L**earning **H**yperheuristic Plus (DRLH$^+$).

In the DRLH$^+$ framework, we replace the ALNS-inspired reward function of DRLH with a new function named $R_{improvement}$. The proposed function uses the percentage improvement from the initial solution as part of the calculation of rewards. This approach sets out to reduce the possibility of achieving large rewards without a corresponding improvement of the objective.

This paper also introduces a hyperheuristic framework that combines the heuristic selection from DRLH$^+$ with the move acceptance of DRLMA, referred to as the **D**ual-Network **D**eep **R**einforcement **L**earning **H**yperheuristic (D^{2}RLH). This framework applies the same intelligent agent for heuristic selection as DRLH$^+$, but adds a second intelligent agent for determining acceptance of found solutions.

1.3 Thesis Outline

The rest of this thesis is organized into the following chapters.

Chapter 2 - Background provides an introduction to subjects that are relevant to the problem and the solution methods. The theoretical background is essential for understanding the functionality of the presented work, and the analysis of its performance.

Chapter 3 - Literature Review covers relevant literature on the PDPTW and use of deep reinforcement learning in hyperheuristics, and provides context on the thesis' place in the literature.

Chapter 4 - Problem Description describes the PDPTW, the problem we test the proposed solution methods on. The chapter describes the objective of the problem, relevant constraints and what a solution to the problem must include.

Chapter 5 - Models presents the two proposed hyperheuristic frameworks. After general details on the search and the available heuristics, the components of both hyperheuristic frameworks are presented.

Chapter 6 - Experimental Setup defines the datasets used for training and testing, and lists selected values for all hyperparameters. Baseline models that represent existing methods are also defined in this chapter.

Chapter 7 - Results presents five conducted experiments and their results. Performances of both frameworks compared to baseline models are discussed here.

Chapter 8 - Conclusion summarizes the work and proposes future work that could further develop the presented methods.

Chapter 2

Background

2.1 Combinatorial Optimization

Optimization is a field of science concerned with finding the best possible decision, given an objective and a set of criteria. Combinatorial optimization is a subfield of optimization which attempts to find an optimum object in a finite collection of objects Schrijver [14]. Combinatorial optimization problems are concerned with discrete objects, with solutions presenting a selection or permutation of variables that should minimize or maximize an objective value.

An intensily studied set of combinatorial problems is the Vehicle Routing Problem (VRP) and it's variations [5]. To solve this problem, a solution must present a collection of routes that can serve demands from customers at different locations. The available vehicles are associated with a capacity that constrains the set of possible routes. The objective is to minimize the cost related to travelling the routes. Solving this problem efficiently is an important field, both in terms of environmental impact and economy.

As selecting a combination of elements is a very general concept, combinatorial optimization problems are relevant to many fields, such as logistics, economics and scheduling. Since many of them become too large for exhaustive search as the scale of the problem increases, the development of methods to efficiently find good solutions is useful to many.

2.2 Heuristic Solution Approaches

Some combinatorial optimization problems, including the VRP and the Travelling Salesman Problem, belong to the NP-Complete complexity class. This is an important distinction, as this set of problems does not have any known polynomial-time algorithmic solutions. If any of the problems that belong to this class is proven to be solvable in polynomial time, it will follow that the same applies to every member of the class, but it is still unknown if that is the case.

Regardless of the possible existence of more efficient solutions to these problems, the solutions currently available to us are exponential in time relative to the size of a particular instance. In many real world use cases, the large number of decision variables turn exact solution methods obsolete, as there might simply not be enough time to wait until an algorithm reaches an optimal answer. Heuristic solution approaches try to meet this demand by finding near-optimal solutions in a much smaller time frame.

Silver [18] provided an overview of heuristic methods, where a heuristic was defined as a method that is likely to produce a reasonable solution on the basis of experience or judgement, but cannot guarantee an optimal solution. These strategies are derived from ideas of what intuitively may produce a good solution, rather than mathematical reasoning. For instance, a heuristic could attempt to optimize one small aspect of the solution, rather than the full solution as a whole.

To fully capitalize on the advantages of heuristics, it is important to tune the balance between *intensification* and *diversification*. Intensification is an attempt to improve a solution as much as possible, which can often lead to quite deterministic functions. Intensifying heuristics that rely on complicated calculations can help with efficiently moving towards local optimums. Conversely, diversifying functions will disregard the likelihood of improvement, and mostly focus on discovering new solutions. Heuristics that incorporate more randomness can ensure that larger areas of the solution space is explored. Being able to diversify in a short time frame is a benefit that is less attainable with exact solutions, especially when dealing with a vast solution space.

2.2.1 Local Search

A local search is a type of heuristic search where the heuristics make changes to an existing solution, as opposed to constructing an entirely new solution in each iteration. The aim of the search is to find local improvements by searching the *neighborhood* of the current solution. Silver [18] defines the neighborhood of a solution as the set of solutions that can be found by applying some simple transformation t. Using VRPs as an example, a simple transformation could remove a single task from a solution, and reinsert it in the placement that minimizes the cost of that task. The new route will still be similar to the previous, but the alteration will probably change the total cost to some degree.

A major weakness of the method is that the result relies heavily on the starting point of the search. If the search succeeds, it yields an approximate local optimum of the solution space. The quality of a local optimum relative to the global optimum can show substantial variation, especially for large problem sizes. For these cases, a simple local search might not exhibit enough diversification to find good solutions.

2.2.2 Metaheuristics

A *metaheuristic* is a problem-specific implementation of an algorithm according to the guidelines of a high-level problem-independent framework [20]. The general framework defines strategies that can be applied to a wide variety of problems, but include some components that have to be adapted to the problem at hand to perform well.

The concept of metaheuristics enables the use of a wide range of methods, usually incorporating both intensification and diversification to achieve good performance. Some examples of well known metaheuristics include Genetic Algorithm, Simulated Annealing and ALNS.

2.2.3 Hyperheuristics

Hyperheuristics are abstractions of the heuristic selection process, made in an attempt to generalize the process to a level that's applicable to multiple problem types. Burke et al. [3] referred to hyperheuristics as search methods for selection or generation of heuristics to

solve a search problem. While metaheuristics depend on problem-specific implementations and search directly in a solution space, hyperheuristics search in a space of heuristics. The intention behind this higher level of abstraction is to develop strategies that are independent of the problem's dynamics.

For a hyperheuristic framework to be applicable to multiple problems, it is important that the experience it learns from is general. If a hyperheuristic selects or generates a heuristic based on the state of the search, the state must be represented by general information that is relevant regardless of the problem type. This method allows the framework to generalize experience regarding search progress independently of the problem type.

Burke et al. [3] classifies hyperheuristics according to the nature of their search space and their learning process. The first dimension separates between *heuristic selection* and *heuristic generation*. These categories can further be classified by the type of heuristics selected or generated, distinguishing between construction heuristics and perturbation heuristics. Construction heuristics follow some procedure to construct new solutions from partial solutions, while perturbative heuristics modify existing solutions that are complete.

The second dimension differentiates between *online learning* and *offline learning*. During online learning, the hyperheuristic learns continuously while solving an instance of a problem. Offline learning relies on generalizing behavior learnt by solving a training set, and applying it to unseen instances.

Both hyperheuristic frameworks presented in this thesis include a component for selection of perturbative heuristics. A deep neural network is trained offline to select heuristics, only relying on general information about the search progress to make decisions.

2.3 Adaptive Large Neighborhood Search

Ropke and Pisinger [13] suggested that many local search heuristics didn't search a large enough area of the solution space, as computationally efficient heuristics made too small changes to a solution. To counteract this issue, they proposed using heuristics that can rearrange a large fraction of the elements in a single iteration, to ensure that the search isn't constrained to a small area of the solution space (large neighborhood). This strategy

was paired with simple insertion heuristics to promote diversification and rely on a more thorough search to discover good solutions.

These heuristics were utilized in an adaptive metaheuristic named Adaptive Large Neighborhood Search (ALNS), which divides the search into segments of n iterations. After every n iterations the heuristics are assigned a weight w, calculated from scores awarded when the use of a heuristic has discovered a new solution, lead to an improved current solution or a new global best. The scores can be adjusted to reflect how important each of these discoveries are regarded.

Heuristics are selected using a roulette wheel selection strategy. From a total of k heuristics, the probability of selecting heuristic j will be

$$\frac{w_j}{\sum_{i=1}^{k} w_i} \tag{2.1}$$

for the entirety of the next segment [13]. This allows the framework to adapt the probabilities according to how well they have performed during the search and prioritize the heuristics that have proved to be the most useful.

2.4 Simulated Annealing

Simulated annealing is a metaheuristic inspired by annealing, a heat treatment that slowly cools a material. Ropke and Pisinger [13] used the acceptance criterion from simulated annealing in their ALNS framework, to decide if a proposed solution should be accepted. Solutions that improve upon the current are automatically accepted, while the criterion handles acceptance of worse solutions. A probability of accepting is calculated based on a temperature parameter T and the difference in objective, with the expression

$$Pr\{accept(l')|l', l, T\} = e^{-\frac{f(l')-f(l)}{T}}, \tag{2.2}$$

where l' is the proposed solution and l is the current solution. The numerator of the exponent is referred to as the *energy difference*. This is the difference in objective value,

assumed to be from a minimization problem, between the proposed and current solutions. A large energy difference implies a much worse solution, and will lead to a low probability of acceptance. A small difference is more tolerable and will have a probability closer to 1.

The temperature can be leveraged to tune the balance between intensification and diversification. By lowering the temperature, the probability can be gradually decreased for the entire length of the search to emphasize diversification at the start and shift towards preferring intensification by the end. The temperature is decayed exponentially by multiplying with the cooling rate $cr \in [0, 1]$ in every iteration. In this thesis, the cooling rate is calculated to decay T to a hyperparameter T_f by the final iteration. After the first 100 iterations of the search, the start temperature is calculated to the value that yields 0.8 probability for the mean energy difference. For this mean, only iterations where the proposed solution was worse than the current are considered. These methods for defining the initial and final temperatures allow us to control the range that the temperature is decayed within.

2.5 Reinforcement Learning

Humans learn by trial and error. We explore the consequences of different actions for ourselves and our environment, and try to balance the options of capitalizing on gained knowledge and exploring new choices. *Reinforcement learning* emulates this behavior by letting an *agent* interact with an *environment*. The agent gains experience by observing how its actions influence the environment around it. To quantify if the changes to the environment conform with our goal, the agent receives a numerical *reward* in response to each action. The main objective is to let the agent learn how to map situations to actions, in order to maximize the received rewards. [19].

The agent's action changes the environment and triggers a reward, and the agent again makes a choice based on the new environment. This circular dynamic can either run continuously or until some form of stopping condition. When a certain number of steps or another type of terminal state T determines when the interaction is over, we call the period from the start of the interaction to the agent is in the terminal state an *episode*. In relation to solving combinatorial optimization problems, an example of one episode can be solving one instance of a problem.

An integral part of being able to apply reinforcement learning to a problem is to have an accurate and formal definition of the agent's goal. The agent's goal is to maximize the *return*, which is the sum of all rewards R the agent receives in the future. For episodic problems, the return at time step t is the sum of all rewards the agent will receive from time step t until the end of the episode. To be able to take into account how important rewards received in the future are relative to immediate rewards, the concept of *discounted return* is needed:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma R_{t+3} + ... = \sum_{k=0}^{T-t} \gamma^k R_{t+1+k}, \tag{2.3}$$

where $(T-t)$ is the number of steps left in the episode, and γ is the *discount rate* bounded by 0 and 1 [19]. When γ is 1, equation (2.1) simplifies to the term for the undiscounted return, but when it is lower, rewards lose value the further ahead in time they will be received. An agent trying to maximize this return will have to make decisions that not only lead to the largest possible reward at the current step, but rather decisions that will lead to paths with larger returns.

2.5.1 Reward Design

The design of the reward function is a vital component of reinforcement learning. It is the formal representation of the objective of the learning process, and serves as a bridge between our intention and the agent's desired behavior. Each reward is intended to be a metric of the agent's success, and therefore must accurately represent what we are trying to achieve.

It is important to be aware of harmful behavior that could occur if the design of the reward function is suboptimal. *Reward hacking* is a behavior describing an agent that successfully increases its rewards in an unintended way. This can occur when the reward function permits some clever solution that formally maximizes it, without adhering to the designer's informal intent [1]. From the agent's point of view, it is an entirely valid strategy that achieves the requested objective. To avoid permitting this type of behavior, the metric of success must be as directly correlated to and representative of the objective as possible.

2.5.2 Agent Policies

A reinforcement learning agent must follow a *policy*. The policy is what defines the behavior of the agent, by explaining how it responds to a given environment. Formally, the policy maps each state to a probability distribution that describes the probability of selecting each action in that state [19]. A policy π can be formulated as

$$\pi(a|s) = Pr\{A_t = a|S_t = s\}, \tag{2.4}$$

where $\pi(a|s)$ is the policy's probability of selecting action a given the state s. There are different methods for calculating the probability distribution. Some policies are defined by some selection strategy after evaluating actions in relation to the current state.

A policy can also be parameterized explicitly by a function. The definition of the policy is extended to

$$\pi(a|s, \theta) = Pr\{A_t = a|S_t = s, \theta_t = \theta\}, \tag{2.5}$$

where the additional input θ is a vector of parameters. A simple type of parameterized function can be a linear function of the parameter vector and a *feature vector* that represents s. As a more advanced example, the feature vector could be fed into a neural network that outputs the probability distribution. With this approach, the parameters θ are the weights of the network.

2.5.3 Value Functions

An agent can use a type of *value-function* to learn a policy. A *state-value function* assigns a value to each state, by estimating the return it expects to receive starting from the respective state. An *action-value function* instead estimates the expected return for each possible action starting from a state, assigning a value to each pair of action and state. The agent uses encountered situations and their subsequent rewards to evaluate the quality of actions

or states, by updating values in the function. As the agent continues to learn, the value-function is readjusted to take the new findings into account. A value-function can implicitly define an agent's policy, by leveraging the function for decision-making. A common type of policy is always selecting the action with the largest expected return in the given state.

For manageable state spaces it is possible to keep a tabular solution that maps specific states to a single value for each action, but for continuous or very large state spaces this method could either be insufficient or even impossible to execute. In such cases, the agent can instead learn by *function approximation*, a method that replaces the one-to-one mapping with an approximation of the entire function. Approximate solution methods allow the learner to generalize knowledge gained from states encountered during training, and apply it to entirely new states that have not been encountered at all.

2.6 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is a combination of deep learning and reinforcement learning where a deep neural network is used as a function approximator in a reinforcement learning setting. While many machine learning methods only rely on one input and one output layer and may require the input to be manually transformed into features before training, deep neural networks have at least one hidden layer between the input and output layers [11]. This feature provides several advantages in relation to reinforcement learning.

The hidden layers of a deep neural network can learn both higher-level and lower-level features directly from raw input [11]. As a result, there is no need to manually engineer features before using the state of an environment as input. A network can approximate a function by mapping the current state directly to the desired value. The layers of a network consist of units that receive a weighted sum of every unit's output from the previous layer. A nonlinear activation function between layers is used to break linearity and ensure that nonlinear functions can also be learned. Given a large enough network and sufficient computational power, any arbitrarily complex behavior can be approximated.

A deep neural network can be used to perform different roles, depending on the reinforcement learning method. The network may be used to approximate value-functions that can implicitly define an agent's policy, such as state-value functions or action-value functions.

In this case, the network would receive a state or a (state, action) pair as input, and output the estimated return associated with it. As the weights are trained, it learns to estimate the value of states and actions more accurately. Alternatively, it can explicitly parameterize a policy by yielding an action or a probability distribution over actions as output. This approach is used to great effect in policy gradient methods.

2.6.1 Policy Gradient Methods

Policy gradient methods are a set of methods that learn a parameterized policy, by applying an approximation of gradient ascent to the parameters. The parameters are updated to maximize the objective $J(\theta_t)$ according to

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)}, \tag{2.6}$$

where α is the learning rate and $\widehat{\nabla J(\theta_t)}$ is an approximation of the gradient of the expected return. The gradient can be approximated in several ways, and methods often rely on the term $\nabla\pi(a|, s, \theta)$ as a result of the Policy Gradient Theorem [19]. This term is the gradient of the probability of taking the action a.

Some policy gradient methods belong to the subclass referred to as *actor-critic methods*. These methods learn an approximation to a value function in addition to the parameterized policy. The value function serves as a *critic* that evaluates the expected return from each state, given that the agent acts according to the current parameterized policy. The policy serves as a *actor* by optimizing the behavior according to the estimated value of the states.

Proximal Policy Optimization (PPO) is an actor-critic method that was proposed by Schulman et al. [16]. The method parameterizes a stochastic policy, producing a probability distribution and not an explicit action. The PPO algorithm improves upon Trust Region Policy Optimization (TRPO) [15], which replaced the objective $J(\theta)$ with a "surrogate objective":

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta old}(a|s)} \tag{2.7}$$

$$J(\theta) = \hat{E}_t[r_t(\theta)\hat{A}_t] \tag{2.8}$$

The first equation is the *probability ratio* $r(\theta)$ between the probabilities of selecting action a before and after updating the policy. The second equation is the new objective, which maximizes the expected value of this ratio multiplied with the advantage function $\hat{A}$. The advantage function estimates the advantage of the selected action relative to the probability distribution of the current policy. It compares the estimated value of the action to the estimated value of the state in general under the policy. Updates increase the probability of selecting actions with positive advantage, and decrease it for actions with negative advantage.

Large updates in favor of an action with a significant advantage can drastically change the policy. PPO introduces a "clipped surrogate objective" that optimizes the minimum of the normal TRPO objective and a version with a clipped probability ratio. A clip parameter ϵ determines the range $[1-\epsilon, 1+\epsilon]$. This leads changes in probability outside of the clipped interval to not improve the objective, rendering them pointless unless they improve the policy in general. Removing the large policy updates stabilizes the learning and justifies training with multiple epochs. The resulting loss function also includes an *entropy coefficient* to ensure sufficient exploration.

The advantage function used in this paper is calculated with Generalized Advantage Estimate (GAE) [17]. The estimation is parameterized by $\gamma \in [0, 1]$ and $\lambda \in [0, 1]$. The discount rate γ determines the weighting of future rewards, as explained in section 2.5. The decay parameter λ decays the advantage estimate to further contribute to the bias-variance trade-off of approximate value functions. The advantage estimator GAE is defined by the expressions

$$\delta_t = (r_t + \gamma V(s_{t+1})) - V(s_t) \tag{2.9}$$

$$\hat{A}_t = \sum_{k=0}^{T-t} (\gamma\lambda)^k \delta_{t+k} \tag{2.10}$$

The δ_t term is the advantage in the 1-step estimate of return compared to current estimated value of the state. $V(s_t)$ represents the estimated value of state s_t. The estimator $\hat{A}_t$

is an exponentially weighted sum of the advantages from each remaining step of the episode. Setting the parameters near 1 leads to high variance and low bias. Lowering the values introduces bias, but decreases variance.

Chapter 3

Literature Review

Variations of the VRP have been studied intensively, but many have also researched the specific variation of PDPTW. Dumas et al. [4] presented an exact algorithm to solve the problem to optimality. Calls are distributed among a subset of the set of all possible routes, formulated as a set partitioning problem. The subproblem is solved repeatedly using a dynamic programming algorithm to provide the subset of routes to the set partitioning problem. The authors emphasize that the algorithm depends on instances to have large demands from customers relative to the vehicles' capacities. In such cases, the capacity constraints restrict the problem to a larger degree and make the solution space more manageable. The load sizes were also simplified to all being one third of the vehicle capacity, half of the capacity or equal to the full capacity. The algorithm was tested on instances ranging from 19 to 55 calls, and found answers with optimality gaps ranging from 0% to 3.2% on all instances.

An early use of reinforcement learning in a hyperheuristic setting was developed by Burke et al. [2]. A tabu-search hyperheuristic was proposed, where heuristics that had not performed well would be considered tabu for some time. Additionally, each heuristic received a score that was updated based on the difference in the objective value after applying the heuristic to the solution. This update was inspired by ideas from reinforcement learning, but in a simplified manner. The scores reflect how much the heuristic has improved the inputs, but consider no general information about the search.

Ropke and Pisinger [13] consider a version of PDPTW where calls are only compatible with a subset of the vehicles, which also applies to the version considered in this thesis.

However, there are no specific costs related to outsourcing a call, and they instead include the minimization of unserved calls as part of the objective function. The proposed ALNS metaheuristic applies sets of removal and insertion heuristics to solve for a set number of iterations. The algorithm selects operators with a roulette wheel selection, utilizing a probability distribution calculated from scores that measure how well the operators have performed in the search in the past iterations. The selection scheme is paired with simulated annealing as acceptance strategy. Ropke and Pisinger [13] found that ALNS performed well in general and improved upon several of the best known solutions at the time of the PDPTW instances that were tested. It worked especially well on larger problem sizes, compared to previously proposed heuristics.

Hemmati and Hvattum [6] evaluated the importance of randomizing different components of the ALNS algorithm by comparing with deterministic alternatives. Results showed that both options had comparable performance. However, across repeated runs on the same instance it appeared that randomized components often lead to larger variance, despite having similar average performance to the deterministic counterpart. This is an advantage when evaluating based on best found solutions, as it will lead to a wider range of solutions after multiple runs. The results also indicated that the randomization of the acceptance function was the main contributor to the increased variance This analysis validates the importance of stochastic acceptance functions like simulated annealing, and including probability in general.

An advantage of involving DRL in a hyperheuristic framework, is that large networks are suited to deal with large amounts of information. This allows the addition of more thorough information on the state of the search and its influence on decision-making than earlier hyperheuristics were able to account for. Lu et al. [12] proposed a framework to let a DRL agent select from a pool of "improvement operators", to iteratively improve a solution. This method was shown to perform well compared to earlier heuristic approaches which were not learning-based. An aspect that could be improved is the state representation used, which is specific to the Capacitated Vehicle Routing Problem (CVRP) and not easily generalizable. Additionally, the framework only uses the agent for selection of heuristics to improve the solution and relies on a separate escape strategy to escape local optima after a set number of iterations without improvement.

A more generalized approach that takes more advantage of DRL can be achieved by allowing the agent more control over decision-making. Kallestad [10] developed an offline

hyperheuristic DRLH that trains an agent to learn to select perturbative heuristics for solving combinatorial optimization problems. A diverse set of heuristics allows the agent to control the balance between intensification and diversification. The agent was trained with PPO on a set of training instances, before being tested on unseen instances. The framework was tested on four different problems, including PDPTW. To keep the hyperheuristic problem-independent, a state representation with only general information about the search progress was used.

DRLH uses a reward function R_{5310}, inspired by ALNS. The agent is rewarded based on the quality of the solution produced by the heuristic it selects. It receives a reward of 1 for discovering a solution that has not been encountered yet in the search, 3 for improving the current solution and 5 if the new solution improves upon the best found solution so far. A potential weakness of this reward function is that it is unaffected by the magnitude of the improvement, and only considers its occurrence. Letting the agent take the size of improvements into consideration, could allow it to assess the quality of each action in more detail.

The PDPTW instances used to test DRLH were from a set of benchmark instances provided by Hemmati et al. [7]. The paper distinguishes between *short sea* and *deep sea* shipping problems, which determines the size of the operational area and distances between nodes. It further separates between *mixed cargo sizes* and *full load cargo sizes*. For the former case, ships can carry multiple cargoes, while the latter has cargoes that amount to a full shipload. By using each combination of these two properties, four sets of 60 instances each were created. Each set provides five instances for each of 12 separate problem sizes. Optimal solutions for 239/240 benchmark instances are available due to contributions by Hemmati et al. [7], Hemmati and Hvattum [6] and Homsi et al. [8]. DRLH was tested on four of the problem sizes from the *short sea + mixed cargo sizes* set.

Isaksen [9] developed a DRL agent named DRLMA for handling acceptance of proposed solutions, as part of a hyperheuristic framework. The framework trains the acceptor agent with PPO, in a comparable setup to DRLH. Only problem-independent information is leveraged in its decision-making. The agent was tested in combination with both ALNS and DRLH as heuristic selector functions. These frameworks were used to solve instances of the Capacitated Vehicle Routing Problem (CVRP) and the Parallel Job Scheduling Problem (PJSP). The inclusion of DRL in the move acceptance was shown to perform well on both problems compared to simulated annealing. DRLMA has not been applied to PDPTW.

This paper aims to improve upon the DRLH and DRLMA frameworks by introducing a new reward function for heuristic selection that is more robust and general, and also combine this new framework with DRLMA. This idea leads to the introduction of the DRLH$^+$ and D^{2}RLH frameworks. We also want to test the ability of both frameworks to solve PDPTW, and thoroughly document how their performances generalize across different sets of PDPTW benchmark instances.

Chapter 4

Problem Description

The problems we solve in this thesis are various sizes of the PDPTW, which is a variant of VRP. In each instance, there is a set of vehicles that is available to serve a set of calls. Each call involves a vehicle picking up cargo at a specific node and delivering the cargo at another node. Figure 4.1 illustrates an instance of the problem, with routes for a proposed solution.

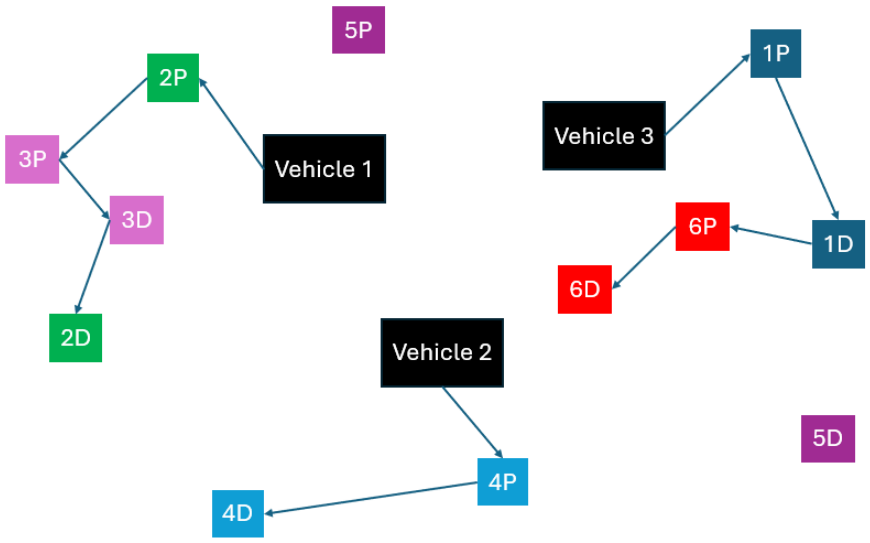


Figure 4.1: Illustration of a PDPTW instance

Illustrates a proposed solution to an instance with 3 available vehicles and 6 calls to be served. Each colored pair of boxes represents a call, with P and D referring to pickup and delivery nodes. Lines indicate a vehicle's route, and any call not included in a route is served by the dummy vehicle.

A solution to an instance of PDPTW is an allocation of calls among the set of vehicles, with a specific permutation of each vehicles route. The pickup and the delivery of a call must be performed by the same vehicle, and all calls must either be outsourced or served exactly once by a vehicle. If we outsource a call, we compensate someone else for handling the call for us at a typically large fee. Outsourced calls are represented by an additional *dummy vehicle*.

A solution is only feasible if it adheres to a set of constraints. The pickup must always be handled before the delivery of the call. There are time windows that specify when each call can be picked up and delivered, which restricts the set of possible permutations for each route. Travel times between nodes and service times at each node, which are specific to each vehicle, determine the earliest time a vehicle can arrive at each node in the route. The vehicle is allowed to arrive before the allowed time window and wait, but arriving after the window makes the solution infeasible. Each vehicle also has a start time, which determines when the vehicle is allowed to leave its origin node and start traveling its route.

There is also a load size associated with the cargo of each call, and the total weight onboard a vehicle is the summed load sizes of all picked-up calls that are yet to be delivered. There is a capacity associated with each vehicle, which the total weight is never allowed to be greater than. Finally, the compatibility constraint associates each call with a subset of vehicles that are compatible with it, implying that any vehicle not appearing in the subset is not allowed to handle the call. The dummy vehicle is always allowed to serve any call. Every instance defines distinct values for travel times, load sizes, capacities and compatibility.

The objective is to create a solution that minimizes the total cost of handling all calls. The main component of the cost is the total travel cost associated with traveling between nodes. The travel cost between every pair of nodes is defined by the instance for each vehicle separately. Additionally, a port cost is related to each pickup and each delivery, which is also separately defined for each vehicle. The final component is the fee for outsourcing calls, with each call having a separate fee. For the full mathematical model of the problem, readers are referred to Hemmati et al. [7].

Chapter 5

Models

In this chapter, our solution methods used to solve instances of PDPTW are presented. The first model utilizes a DRL agent, tasked with selecting a heuristic in each iteration. The proposed model is referred to as the **D**eep **R**einforcement **L**earning **H**yperheuristic Plus (DRLH$^+$), designed with an emphasis on improving the reward function of the heuristic selector agent.

The second model reuses the same agent for heuristic selection, but adds a second DRL agent to the framework. By determining if a proposed solution should be accepted or not, the new agent replaces the acceptance function of the framework. Given the dynamic of two agents' networks each serving different roles in the search process, the model will be referred to as the **D**ual-Network **D**eep **R**einforcement **L**earning **H**yperheuristic (D^{2}RLH).

Before the hyperheuristic frameworks of the models are presented, including descriptions of both agents, some details on the search method are specified.

5.1 General Search Method

This section describes the nature of the search in general. These details remain the same when solving an instance of the problem, regardless of the model and whether the agents are being trained or tested.

5.1.1 Search Length

The models have a maximum of 1000 iterations to find a solution to an instance of the problem, unless otherwise specified. In each iteration, a new solution is produced by a heuristic after receiving the current solution as input. The acceptance function then determines if the proposed solution will replace the current solution or not. It follows that a heuristic will be applied 1000 times during the search, and the solution encountered with minimum cost is the one provided by the algorithm.

5.1.2 Solution Representation and Initial Solution

A solution to the problem is represented by a list of vehicles. All vehicles are represented by a single list, with an additional dummy vehicle serving all calls that are outsourced. Every call must be served by exactly one vehicle, appearing twice to include both pickup and delivery by the same vehicle. The first time a call appears in a list always represents the pickup, and the second represents the delivery. With this representation, each solution specifies both a distribution of calls among available vehicles and a specific permutation of each vehicle. Figure 5.1 illustrates the representation of a solution.

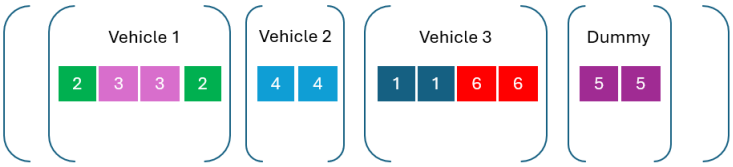


Figure 5.1: Solution representation for PDPTW exemplified

Illustrates the solution representation of the solution displayed in figure 4.1. Each list represents the calls served by a vehicle, with the permutation defining the order of the route. Each colored pair of boxes represents a call, with the first and second boxes representing pickup and delivery, respectively.

The initial solution of an instance is always generated by placing all calls in the dummy vehicle. Outsourcing a call instead of serving it with the fleet of vehicles is generally very expensive, leading the cost of the initial solution to be large. This provides a consistent starting point for every search, removing the possibility of unintentionally generating a good solution in advance.

5.1.3 Available Heuristics

In each search, the same set of heuristics H is available to the heuristic selector. The set consists of 29 heuristics, where all except one is a removal operator paired with an insertion operator. The sizes of the sets of removal and insertion operators are $|R| = 7$ and $|I| = 4$, totaling 7 x 4 = 28 possible heuristics. The only one that doesn't conform to this format is an independent heuristic, which is added to complete the set H. The set matches the heuristics provided by Kallestad [10] when testing their DRLH model.

When applying a combined heuristic, a removal operator first selects n calls to be removed from the solution, without altering the order of the remaining calls. The following insertion operator reinserts all calls that were removed by following some principle. The additional heuristic is the only one that follows one coherent strategy, as opposed to being separated into individual removal and insertion operations.

All of the heuristics follow problem-independent strategies and could be applied to a wide range of problems, but have in our case been implemented to comply with our solution representation of PDPTW. The removal operators, the insertion operators and the additional heuristic are described in more detail below.

Removal Operators

The set R consists of 7 removal operators. While all the insertion operators are deterministic to some degree, a subset of the removal operators is the main source of diversification in the search process. The first five operators in table 5.1 remove a completely random subset of calls from the solution. These operators only differ in the number of calls they remove, as they randomly select a number n from their respective ranges. Multiple sizes are included to be able to adjust the degree of movement in the solution space for each iteration.

The operator $remove\_largest\_\Delta C$ orders calls according to how much the cost of the current solution will decrease if the call is removed. This metric named ΔC only evaluates the cost difference when not serving the call at all, without considering the cost of reinserting it. The operator selects a random number n in the range 5-10, and the n calls with largest ΔC are removed from the solution. This strategy intends to identify calls with inadequate

placements. A potential drawback is that the operator becomes more deterministic for instances where certain calls are expensive in general, which could lead to the repeated selection of certain calls regardless of their placement in the solution.

Operator name	**Functionality**
Remove_random_xs	Removes a random subset of 2 to 5 calls
Remove_random_s	Removes a random subset of 5 to 10 calls
Remove_random_m	Removes a random subset of 10 to 20 calls
Remove_random_l	Removes a random subset of 20 to 30 calls
Remove_random_xl	Removes a random subset of 30 to 40 calls
*Remove_largest_*ΔC	Removes the top 5 to 10 calls with largest decreased cost
Remove_consecutive	Removes a random segment of 2 to 5 consecutive elements

Table 5.1: List of available removal operators in the set R

The last removal strategy is *remove_consecutive*, which removes a segment of 2-5 consecutive elements from the solution. The elements are either a pickup or a delivery, and not a complete call. However, all calls that lose either of these elements will be fully removed, which means that any number of calls between 1 and 5 can potentially be removed. This operator targets the specific permutation of a solution, providing an opportunity to rearrange them in new orders.

Insertion Operators

The set I consists of 4 insertion operators, and all of them incorporate an intensifying aspect. The sizes of the insertion operators are undefined, and instead depend on the removal operators they are paired with. All calls must be served exactly once to compose a valid solution, and as a consequence the insertion operator must reinsert the same number of calls n that were removed by the preceding removal operator. The removed calls are shuffled after all removal operations, and are reinserted in the resulting randomized order unless the insertion operator explicitly reorders the calls. The insertion operators are listed in table 5.2.

The operator $insert\_first$ is a simple insertion method that selects the first feasible insertion that is found. It iterates through a call's compatible vehicles in a random order

and places the call in the first feasible placement into a vehicle it encounters. This procedure is performed until all calls have been reinserted into the solution. *Insert_greedy* iterates through calls in a similar fashion, but compares all feasible placements of a call before greedily selecting the candidate with the lowest increased solution cost.

Operator name	**Functionality**
Insert_first	Inserts calls into the first feasible position encountered
Insert_greedy	Inserts calls into their respective lowest possible cost positions
Insert_beam_search	Finds the best positions to place each call using beam search, and keeps a beam width of 10 after each iteration
Insert_by_variance	Inserts each call greedily, in order of decreasing variance

Table 5.2: List of available insertion operators in the set I

The *insert_beam_search* operator performs a beam search of width 10, by placing 1 call into the solution in each iteration. The operator finds the 5 best placements for a call into each of the 10 solutions from the last iteration, and keeps the 10 best out of the new solutions for the next iteration. After the last call has been added, the best of the resulting solutions is selected.

The last operator, named *insert_by_variance*, is based on the idea that calls that are reinserted earlier have more placements as options available to them. All placements of a call are ordered by lowest increase in cost when reinserting, and the variance of the 10 best options is calculated. Calls are reinserted in order of highest variance, to prioritize the calls that potentially could suffer more in quality from losing their optimal insertions.

Additional Heuristic

The only heuristic that isn't a combination of two operators is *Find_single_best*. This heuristic makes a small change to the solution as it moves a single call. To compensate for the lack of diversification, it finds the most intensifying change that can be made in this way. It iterates through every call separately, and compares the solutions created by moving the call into every other valid position. The best solution obtained by moving each call becomes a candidate solution, and the heuristic selects the solution with the lowest cost of all examined options.

5.2 Hyperheuristic Frameworks

This chapter presents the hyperheuristic frameworks of the two models. During training of the agents, each instance of the problem is regarded as an episode. The DRLH$^+$ and D^{2}RLH algorithms presented in sections 5.2.1 and 5.2.2 are used repeatedly to solve individual episodes from the training set, and the relevant agent is trained according to algorithm 1 based on the rewards received during an episode. These trained networks are then used by the models' respective algorithms when solving new instances during testing.

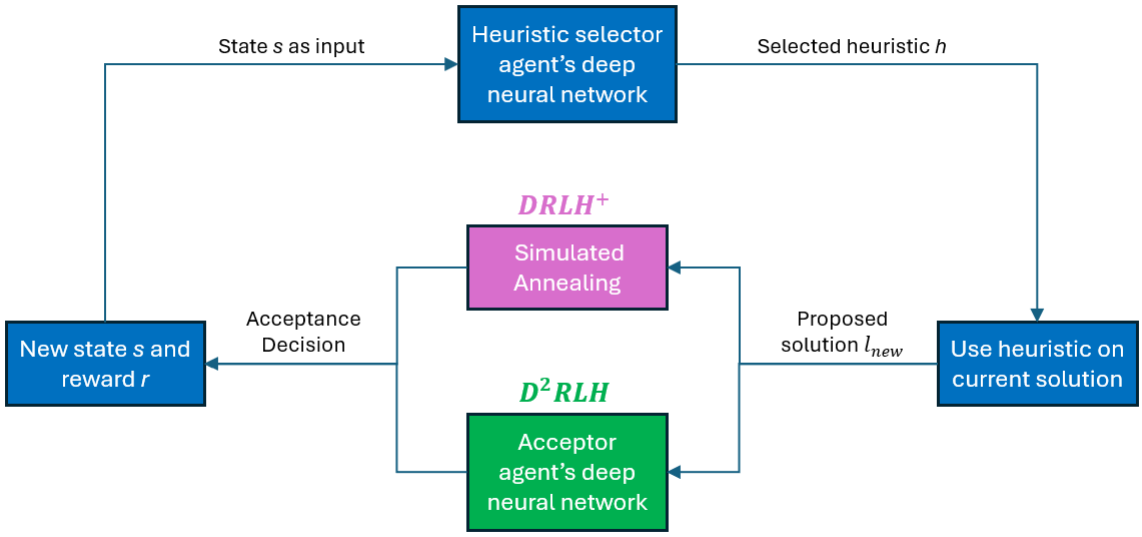


Figure 5.2: Hyperheuristic framework iteration loop

Illustrates the interaction between components of the framework during an iteration.
The acceptance functions are shown in different colors to highlight the difference between the models.
The weights of the heuristic selector (for DRLH$^+$) or acceptor (for D^{2}RLH)
are trained after solving an instance for 1000 iterations.

Figure 5.2 illustrates how the models iteratively find new solutions, and decide if they should be accepted or discarded. It is important to note that the heuristic selector is trained by DRLH$^+$, while D^{2}RLH applies the fully trained selector when training its acceptor agent.

Algorithm 1 General Framework for Training with PPO

Inputs: Hyperheuristic Framework *hf()*, Number of instances *max_episodes*
function TRAIN_AGENT
    Initialize random weights θ
    for $episode = 1, 2, \ldots, max\_episodes$ **do**
        Solve *episode* with $hf()$
        Calculate advantages from agent's rewards
        Update θ with PPO
    end for
    return Trained weights θ for the agent policy π
end function

5.2.1 DRLH$^+$

The DRLH$^+$ model applies a deep neural network as the policy π of a DRL agent. This agent will be referred to as the *heuristic selector agent*. Algorithm 2 describes the procedure of solving an instance by repeatedly using the policy to select a member from the set of available heuristics. The solution produced by the heuristic is automatically accepted if its cost improves upon the previous solution, and the acceptance decision is determined by simulated annealing if not.

It is important to specify details surrounding the training and function of the heuristic selector agent, such as how we represent the state of the search as input for the network, the action space of the agent and the reward function that evaluates its actions. This section defines these components.

Heuristic Selector Agent

The DRLH$^+$ model uses the parameterized policy π of a DRL agent as it's heuristic selection function. The policy defines the probabilities of selecting each action as

$$\pi(h|s, \theta) = Pr\{A_t = h|S_t = s, \theta_t = \theta\}, \tag{5.1}$$

Algorithm 2 DRLH$^+$

Inputs: Heuristic selection policy π, Number of iterations *max_iterations*
function HYPERHEURISTIC
    Generate initial solution l_{curr}
    $l_{best} = l_{curr}$
    for $iteration = 1, 2, \ldots, max\_iterations$ **do**
        Select heuristic h according to $\pi(s, \theta)$
        $l_{new} = h(l_{curr})$
        if $cost(l_{new}) < cost(l_{best})$ **then**
            $l_{best} = l_{new}$, $l_{curr} = l_{new}$
        else if $cost(l_{new}) < cost(l_{curr})$ **then**
            $l_{curr} = l_{new}$
        else if $accept()$ **then**
            $l_{curr} = l_{new}$
        end if
    end for
    return $l_{best}, cost(l_{best})$
end function

where h is a heuristic from H and s is the current state of the search. θ is in this context the current weights of the deep neural network. The agent is trained using PPO (described in section 2.6.1), and the networks have alternating linear layers and ReLU activation functions. In each iteration, the state is passed through the network, and a probability distribution over all available heuristics is received as output. The next section describes what information is included in the state representation.

State Representation

For the agent to make efficient decisions, it is essential that it receives relevant information relating to the state of the search. The included features are presented in table 5.3, and are all descriptive of the nature of the search. The features used by Kallestad [10] are extensive and problem-independent, and applying the same set ensures a consistent baseline for comparing the models.

Feature name	Description
Normalized:	
cost	The cost of l_{curr}
min_cost	The cost of l_{best}
cost_from_min	The difference $cost(l_{curr}) - cost(l_{best})$
reduced_cost	The difference between $cost(l_{curr})$ and cost of the previous solution
index	The current iteration number
T	The current temperature (for Simulated annealing)
cr	The cooling rate (for Simulated annealing)
no_improvement	Number of consecutive iterations without improvement
Not normalized:	
unseen	Binary which is 1 if this is the first time this solution is found
was_changed	Binary which is 1 if current solution changed in the last iteration
last_action	The last action by the selector as a vector in 1-hot encoding
last_action_sign	Binary which is 1 if the last action improved the current solution

Table 5.3: Features included in the state representation of DRLH$^+$

Features such as *index*, *T* and *cr* provide general information on the progress of the search. It is important for the agent to let search progress influence the willingness to diversify or intensify, as it often is beneficial to search for a local optimum towards the end.

Other features are important to be able to gauge the quality of the current solution, both in terms of absolute value and relative to the best solution found so far. The features *cost*, *min_cost* and *cost_from_min* are included to let the agent relate decisions to these values. Additionally, knowing if the solution has been encountered before could indicate what type of area of the solution space we are in, which justifies including *unseen*.

Despite there being some degree of randomness present in most of the available heuristics, it is rarely beneficial to remain in an area for an extended period if there's an evident lack of progress. The agent should have the opportunity to recognize these situations, and the *no_improvement* feature is included for this purpose.

The remaining features are *was_changed*, *reduced_cost*, *last_action* and *last_action_sign*. The erratic nature of heuristic searches causes the suitability of different heuristics to depend

heavily on the situation. These features provide context on the efficiency of the last action in this area of the solution space, which is important to determine when repeating an action could prove helpful. The *reduced_cost* feature could also be useful in recognizing heuristics that are suited to succeed a large or small change in the current solution.

The normalized section of the state vector is normalized by the last 100,000 encountered states, while the other features remain unaltered. The resulting vector is the state representation that the deep neural network receives.

Action Space

The task of the heuristic selector agent is only to select among the set of heuristics, with each specific heuristic being regarded as one action. In this case, where $|H| = 29$, this makes a set of 29 possible actions.

Reward Function

When solving problems with reinforcement learning, the design of the reward function is essential in encouraging the wanted behavior from the agent. When Kallestad [10] proposed the DRLH model, its reward function R_{5310} was inspired by ALNS and gave rewards when improving upon the global best or the current solution. This reward system encourages improvements in general, and includes diversification through a small additional reward for discovering new solutions.

Even though the point-based reward system works well with ALNS, there are some potential pitfalls when the agent is intelligent. Although it is encouraged, the reward system only considers the occurrence of improvement, regardless of the extent. It follows that the optimal policy of the agent maximizes the number of improvements. As a consequence, an intelligent agent could learn to deliberately make improvements as small as possible, which makes it more likely to receive the reward many times. Anecdotal evidence from the training of DRLH supports this theory, where the received ALNS score increased over time without any significant difference to the best solutions. We also observed from training statistics that heuristics that made smaller changes to the solution were selected more frequently.

The maximization of the number of improvements does not always translate well enough to our real objective of finding a solution with minimal cost. Instead, we want to relate the rewards as directly and simply as possible to our objective, in an attempt to minimize the risk of reward hacking. We propose a function $R_{improvement}$, defined as

$$R_{improvement} = \alpha * \frac{cost(l_{initial}) - cost(l_{proposed})}{cost(l_{initial})} + \beta * unseen, \tag{5.2}$$

where α and β are scaling factors to weight the importance of improvement and finding new solutions respectively. The *unseen* term is a binary which is 1 if this is the first time the proposed solution is encountered in the search, which then triggers the reward β. This reward system calculates the reward from the percentage improvement from the initial solution. This should not only encourage constant progress, but also to be as efficient as possible from the start of the search. Any reduction that is maintained will be rewarded in every iteration for the rest of the search. Our formulation of the function assumes that the problem at hand is a minimization problem, but it can easily be repurposed for maximization problems by setting a negative value for α.

The intuition behind applying the initial solution as a baseline is to establish an absolute baseline for each instance. The baseline is independent of the agent's actions, as opposed to values such as l_{best} and $l_{current}$ which depend on the agent's own performance. Additionally, we have added the last term to be able to adjust the degree of diversification. If the solution is near a local optimum, a large enough β value could allow the agent to find worse solutions and escape the area without being punished, as long as it discovers new solutions.

5.2.2 D^{2}RLH

The D^{2}RLH model is similar to DRLH$^+$, with the addition of a second agent to handle acceptance of proposed solutions. While the DRLH$^+$ model used an agent's policy for heuristic selection and simulated annealing for acceptance, this network has policies π for both purposes. Algorithm 3 describes the procedure for solving an instance with this model.

When the acceptor agent is training, a *stationary* selector agent is employed to select heuristics, which is the neural network of the trained DRLH$^+$ model. In this context, a

Algorithm 3 D^{2}RLH

Inputs: Heuristic selection policy π_{select}, Acceptance policy π_{accept}, Number of iterations $max\_iterations$
function HYPERHEURISTIC
    Generate initial solution l_{curr}
    $l_{best} = l_{curr}$
    for $iteration = 1, 2, \ldots, max\_iterations$ **do**
        Select heuristic h according to $\pi_{select}(s, \theta)$
        $l_{new} = h(l_{curr})$
        if $cost(l_{new}) < cost(l_{best})$ **then**
            $l_{best} = l_{new}$, $l_{curr} = l_{new}$
        else if $cost(l_{new}) < cost(l_{curr})$ **then**
            $l_{curr} = l_{new}$
        else if $\pi_{accept}(s, \theta)$ **then**
            $l_{curr} = l_{new}$
        end if
    end for
    return $l_{best}, cost(l_{best})$
end function

stationary agent is one that is fully trained and not updating its network weights after solving problems. This strategy was employed by Isaksen [9], and allows the acceptor agent to learn to behave in relation to the behavior of the selector agent, without turning the hyperheuristic into a multi-agent system. When the acceptor agent also is trained, both agents are kept stationary and used in the framework to solve test instances.

This section specifies the state representation, action space and reward function applied for the acceptor agent.

Acceptor Agent

The D^{2}RLH model's second DRL agent functions as its acceptance function, and adheres to the same principles as the heuristic selector agent. Any proposed solution with a lower cost

than the current solution is automatically accepted. For all other cases, the policy defines the action probabilities as

$$\pi(a|s,\theta) = Pr\{A_t = a|S_t = s, \theta_t = \theta\}, \tag{5.3}$$

where a is the action of either accepting or declining a proposed solution. The state s received by the network is a slightly modified version of the representation presented in section 5.2.1, and is detailed in the section below. The reward system also differs to reflect the role of this agent, while the weights θ are still trained according to algorithm 1.

State Representation

The set of features included in the acceptor agent's state representation has some modifications compared to the version used for the heuristic selector. A vital difference is that the acceptor is applied after the heuristic selector has proposed a new solution that is yet to be accepted or rejected. The new additions try to reflect this distinction by including information that is relevant to the new situation. To make the model comparable to DRLMA, we have provided the agent with the same set of features as presented by [9]. Table 5.4 describes all the features included in the state representation.

Some of the added features measure the proposed solution, which is essential to evaluate its quality when making a decision. This applies to the features named *proposed_cost*, *proposed_cost_from_min*, *improvement* and ΔE. The first two features are comparable to *cost* and *cost_from_min*, with the role of the current solution replaced by the proposed solution. ΔE calculates the difference in cost between the two, while *improvement* is a binary that declares if the proposed solution improves upon the current solution or not.

The range of solution costs can differ remarkably between instances. To provide a metric that is relative to the results so far, the features *proposed_ranks_difference* and *proposed_ranks* are applied. Both are vectors of 3 numbers, describing the rank of the proposed solution when competing with the rest of the solutions encountered in the search. The first entry ranks the solution among all seen solutions, the second among all accepted solutions, and the last among all solutions that were the best solution found when they were first

Feature name	Description
Normalized:	
cost	The cost of l_{curr}
min_cost	The cost of l_{best}
$cost\_from\_min$	The difference $cost(l_{curr}) - cost(l_{best})$
proposed_cost	The cost of $l_{proposed}$
$proposed\_cost\_from\_min$	The difference $cost(l_{proposed}) - cost(l_{best})$
ΔE	The energy difference $cost(l_{proposed}) - cost(l_{current})$
no_improvement	Number of consecutive iterations without improvement
steps_with_current	Number of consecutive iterations since current solution was changed
Not normalized:	
index	The current iteration as a fraction of *max_iterations*
T	The current temperature as a fraction of the start temperature (from Simulated annealing)
improvement	Binary, which is 1 if $cost(l_{proposed}) < cost(l_{current})$
last_action	The last action by the acceptor as a binary
selector_action	The selected heuristic as a vector in 1-hot encoding
proposed_ranks	Solution ranks of the proposed solution
$proposed\_ranks\_difference$	Rank differences of proposed and current solution

Table 5.4: Features included in the state representation of D^{2}RLH

encountered. While *proposed_ranks* contains the absolute ranks of the proposed solution, $proposed\_ranks\_difference$ is the difference in ranks compared to the current solution.

The last features add more information about the recent progress of the search. *steps_with_current* is the number of iterations since the last time the current solution was changed. The last actions of the selector agent and the acceptor agent are described respectively by *selector_action* and *last_action*.

Action Space

The action space of the acceptor agent is quite simple. Since it only decides whether a solution should be accepted or not, the only possible actions are accepting and declining.

This makes it possible to represent the action as a binary variable.

Reward Function

The reward function for the acceptor agent focuses on improvement in the near future by assessing the results η steps ahead of the current iteration. The intention is to accept solutions that lead to better paths of future solutions, and not solely based on the result in the next step. Therefore, the set of solutions found in the next η iterations is viewed as the result of an acceptance decision. We utilize a version of the reward function $R_{\eta \text{ steps ahead}}$ first proposed by Isaksen [9] in their DRLMA model, defined as

$$R_{\eta \text{ steps ahead}} = \begin{cases} \omega_{\text{best}}, & \text{if} \quad cost(l^{\eta}_{best}) < cost(l_{best}) \\ \omega_{\text{improvement}}, & \text{if} \quad cost(l^{\eta}_{best}) < cost(l_{current}) \\ \omega_{\text{else}}, & \text{if neither improved after } \eta \text{ steps} \\ \omega_{\text{accepted without improvement}}, & \text{if} \quad cost(l_{current}) \leq cost(l^{\eta}_{best}) \text{ and accepted} \end{cases} \tag{5.4}$$

In this function, l^{η} denotes the set of the η next proposed solutions found after step t, regardless of their subsequent acceptance decisions. The first three terms are rewards given if the best solution found in the next steps either improves upon the best solution found until step t, improves upon the current solution in step t, or neither of the mentioned cases. Only the first of these three rewards that applies will trigger a reward. The last term is independent of the rest and is a negative reward that is given if the proposed solution in step t is accepted and none of the solutions in l^{η} improves upon the current solution. This penalty encourages to only accept worse solutions if they lead to the discovery of better solutions within a short time frame.

Chapter 6

Experimental Setup

6.1 Datasets

Separate datasets of PDPTW instances are used for training and testing of the models. For each model type, an individual model is trained for each problem size. In addition to training and test sets of corresponding sizes, a more extensive set of benchmark instances is used for additional testing.

6.1.1 Training and Test Sets

All models have been trained on a specific set of training instances. The instances were generated by Kallestad [10], using the instance generator provided by Hemmati et al. [7]. As defined in chapter 3, all of the training instances are of the type *short sea + mixed cargo sizes*. For each model type, one model is trained on each problem size, with a training set of 5000 instances available for each problem size listed in table 6.1.

For each training set, there is a separate test set of 100 instances of corresponding problem size. These instances are also of the type *short sea + mixed cargo sizes*. All experiments are performed on these test sets, with the sole exception of the benchmark tests in section 7.4.

# Calls	# Vehicles	Available training instances	Test instances
18	5	5000	100
35	7	5000	100
80	20	5000	100
130	40	5000	100

Table 6.1: Training sets available

6.1.2 Benchmark Instances

There is an extensive set of benchmark instances provided by Hemmati et al. [7], as described in chapter 3. Optimal solutions are known due to contributions by Hemmati et al. [7], Hemmati and Hvattum [6] and Homsi et al. [8].

# Calls	# Vehicles	# Short Sea Instances	# Deep Sea Instances
7	3	5	5
10	3	5	5
15	4	5	5
18	5	5	5
22	6	5	5
23	13	5	5
30	6	5	5
35	7	5	5
60	13	5	5
80	20	5	5
100	30	5	5
130	40	5	5

Table 6.2: Problem sizes for benchmark instances with mixed cargo sizes

Although all models are trained on *short sea + mixed cargo sizes* instances, we perform an experiment on all four sets to see how well the learnt behavior generalizes to other types of instances. The other sets are "*short sea + full load cargo sizes*", "*deep sea + mixed cargo sizes*" and "*deep sea + full load cargo sizes*". Each set has 60 instances, consisting of 12 different problem sizes with 5 instances each. The problem sizes for the two sets with mixed

cargo sizes are listed in table 6.2, and problem sizes for the sets with full load cargo sizes are listed in table 6.3.

# Calls	# Vehicles	# Short Sea Instances	# Deep Sea Instances
8	3	5	5
11	4	5	5
13	5	5	5
16	6	5	5
17	13	5	5
20	6	5	5
25	7	5	5
35	13	5	5
50	20	5	5
70	30	5	5
90	40	5	5
100	50	5	5

Table 6.3: Problem sizes for benchmark instances with full load cargo sizes

6.2 Hyperparameters

This section describes the hyperparameters used to train the agents of DRLH$^+$ and D^{2}RLH. The heuristic selector agent from DRLH$^+$ is used by both frameworks, which means that its hyperparameters are relevant to both.

Hyperheuristics aim to generalize well to multiple problem types by being problem-independent. Most of the general hyperparameters are selected to be comparable to the approaches by Kallestad [10] and Isaksen [9], to avoid over-tuning components that should not depend on the specific problem. The hyperparameters are presented for both of the proposed models in table 6.4.

The discount rate γ was set to 0.90 for DRLH$^+$, as we believe it should be able to plan for more than the immediate future. However, we also expect a long sequence of solutions, made by heuristics that include randomness, to have a large variance. The selected value is

intended to allow it to recognize longer patterns, without experiencing too much variance to be able to learn. Although an experiment that analyses the effect of the discount rate is included, this was conducted after the hyperparameter was selected for the main experiments.

Hyperparameter	$DRLH^+$	D^2RLH
Epochs	10	10
Learning rate	1e-5	1e-5
Batch size	64	64
Hidden layer sizes	[256, 256]	[256, 256]
Clip parameter ϵ	0.2	0.2
Discount rate γ	0.9	0.99
Decay parameter λ	0.95	0.95
Entropy coefficient	0.01	0.01
Weight decay	0.0	1e-4
KL divergence limit	0.02	0.02
Simulated annealing: warmup phase	yes	no
Simulated annealing: T_f	0.05	0.05

Table 6.4: General hyperparameters for the proposed models

Each trained model has been trained by solving instances from the sets described in table 6.1. For each model type, one model has been trained for each problem size. Due to extensive training times, none of them are trained on the full set. It was observed during training of the models that there were rarely significant changes in behavior after a few hundred episodes, with the current selection of hyperparameters. To balance training time and quality, the training was stopped when the behavior appeared to have comfortably converged. Further research on extended training with a tuned set of hyperparameters would be a useful continuation of this work, given no practical concerns about training time. Table 6.5 describes the number of episodes each model is trained on.

Model Name	**18 calls**	**35 calls**	**80 calls**	**130 calls**
$DRLH^+$	1000	1000	1000	500
D^2RLH	1000	1000	1000	500

Table 6.5: Number of episodes trained on for each trained model

6.2.1 Reward Functions

In addition to the hyperparameters used to train the networks, the adjustable parameters of the reward functions have to be set. $R_{improvement}$ calculates the percentage improvement from the initial solution, which leads to small values as rewards. Additionally, the often unknown optimal solution adds an upper bound to this component, which further restricts the size of the rewards. The parameter α allows us to scale the rewards to a reasonable range, and is set to 2 as shown in table 6.6. β provides a small reward for finding new solutions, which is intended to encourage diversification when improvement becomes limited, and is set to 0.1.

Hyperparameter	$DRLH^+$
α	2
β	0.1

Table 6.6: Hyperparameters of $R_{improvement}$

The parameters of $R_{\eta \text{ steps ahead}}$ determine the range of solutions we consider when rewarding an acceptance decision, and the scale of the rewards. The selected values in table 6.7 were observed by Isaksen [9] to work well for both CVRP and Parallel Job Scheduling Problem (PJSP), and we utilize the same rewards to see how well they work for PDPTW with our new solution methods.

Hyperparameter	D^2RLH
η	7
ω_{best}	3
$\omega_{\text{improvement}}$	1
ω_{else}	0
$\omega_{\text{accepted without improvement}}$	-3

Table 6.7: Hyperparameters of $R_{\eta \text{ steps ahead}}$

6.3 Baseline Models

Some additional models are used for testing, to provide a useful reference for performance. All experiments compare performances to Uniform Random Selection (URS) and ALNS,

with most tests calculating the other models' performances as a percentage improvement from the best costs achieved by URS. A trained DRLH model is also included in relevant experiments. To provide an equal basis for all models, all of them adhere to the general search method described in section 5.1.

6.3.1 Uniform Random Sampling (URS)

URS is a model that selects heuristics from a uniform probability distribution, giving all heuristics an equal chance of being selected in every iteration. This baseline provides a comparison that any helpful strategy should outperform, as strategies that are worse than random selection are not worthwhile. The random selection strategy is paired with simulated annealing for acceptance.

6.3.2 Adaptive Large Neighborhood Search (ALNS)

An ALNS model is included as it is a widely applied hyperheuristic that has no DRL components. This allows us to analyse how well the addition of DRL contributes to the quality of the framework, compared to a well-established strategy. The heuristic selection strategy of ALNS is paired with simulated annealing for acceptance.

The ALNS model runs with a segment size of 50 iterations. A reaction factor $rf \in [0, 1]$ determines how much the scores of the last segment affect a heuristic's new weight. All weights are equal in the first segment, and the weight in segment i is calculated according to equation 6.1 for all heuristics that were used in the segment. s is the heuristic's total score in a segment, and u is the number of times the heuristic was used.

$$w_i = (1 - rf) * w_{i-1} + rf * \frac{s_{i-1}}{u_{i-1}} \tag{6.1}$$

6.3.3 Deep Reinforcement Learning Hyperheuristic (DRLH)

Our proposed DRLH$^+$ model attempts to improve upon DRLH by utilizing the $R_{improvement}$ reward function. To document the advantage gained by this component, some experiments include a DRLH model that uses its R_{5310} reward function. To provide a legitimate comparison, all other hyperparameters are identical to the ones applied to DRLH$^+$, including the number of episodes trained on for each problem size.

Chapter 7

Results

7.1 Experiment on Test Sets

The first experiment compares the performance of the proposed models to the baselines that have no DRL components. For each instance, a model's average cost across 5 runs with different seeds is calculated. The percentage cost reduction achieved by ALNS, DRLH$^+$ and D^{2}RLH compared to the average solution found by URS is averaged over all 100 instances. Figure 7.1 displays these results for each of the four problem sizes listed in table 6.1.

The results show a clear advantage gained from including intelligent decision-making. The performance gap between ALNS and the DRL models expands as the scale of the problem increases. The difference in improvement is substantial for the larger problem sizes, but less evident for the smallest size. This is expected as it is more likely to find near-optimal solutions on very small problem sizes, leaving less room for improvement. It is also notable that DRLH$^+$ has a clear advantage over D^{2}RLH on the larger problem sizes, while D^{2}RLH performs better on the smallest.

The intelligent acceptance of D^{2}RLH appears to learn that acceptance rarely leads to improvements. Statistics from testing shown in figure A.1 show that the model accepts worse solutions much more conservatively than the counterpart that uses simulated annealing. For nearly all 100 instances of the largest problem size, worse solutions are not accepted at all. This complete lack of diversification explains the reduced performance, and the

improvement over ALNS is likely due to the heuristic selection it shares with DRLH$^+$. On the smallest problem size the moderate acceptance appears to achieve a better balance between intensification and diversification, and manages to contribute to the performance.

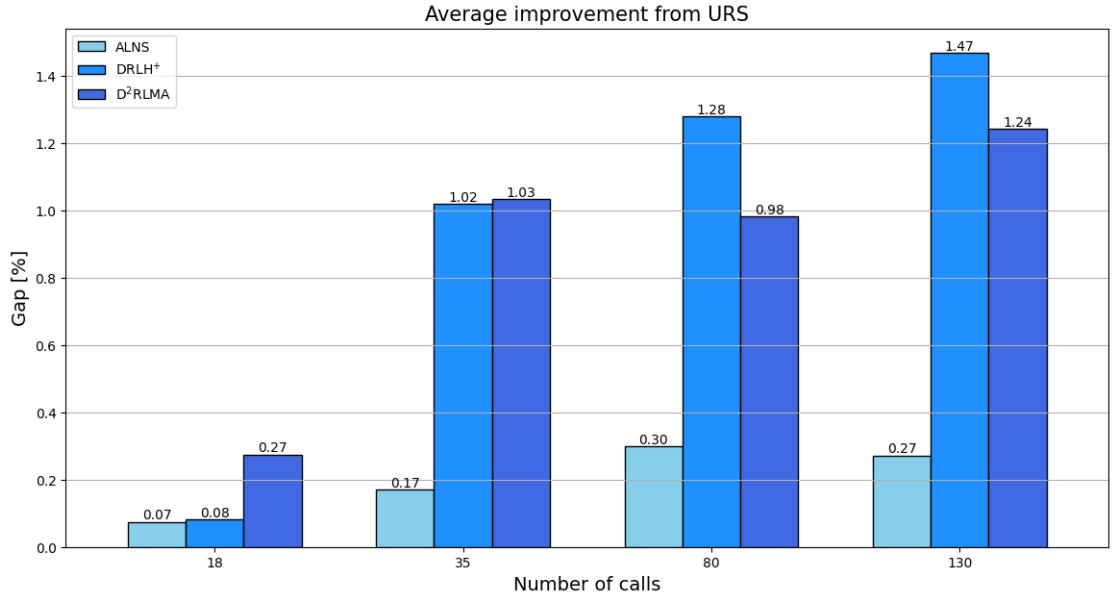


Figure 7.1: Model performances on test sets

The percentage improvement from the average cost found by URS, comparing the proposed models to ALNS on each problem size. Average costs found after solving with 5 different seeds are compared to URS, and improvements are averaged over the 100 test instances.

7.2 Experiment Comparing Reward Systems

The DRLH baseline described in section 6.3.3 is also tested on the same test sets. Comparing DRLH and DRLH$^+$ models with identical configuration allows us to emphasize the difference in performance when replacing the reward function R_{5310} with $R_{improvement}$. Figure 7.2 compares the models' improvement from URS, with ALNS added as an additional baseline.

The new reward function consistently outperforms its predecessor on all sizes except 18 calls. The largest gap in performances is seen on 130 calls, with DRLH$^+$ achieving markedly better performance. The argument behind replacing R_{5310} was its susceptibility to reward hacking, which is supported by statistics shown in figure B.1. The performances indicate that $R_{improvement}$ more successfully represents the objective of the search.

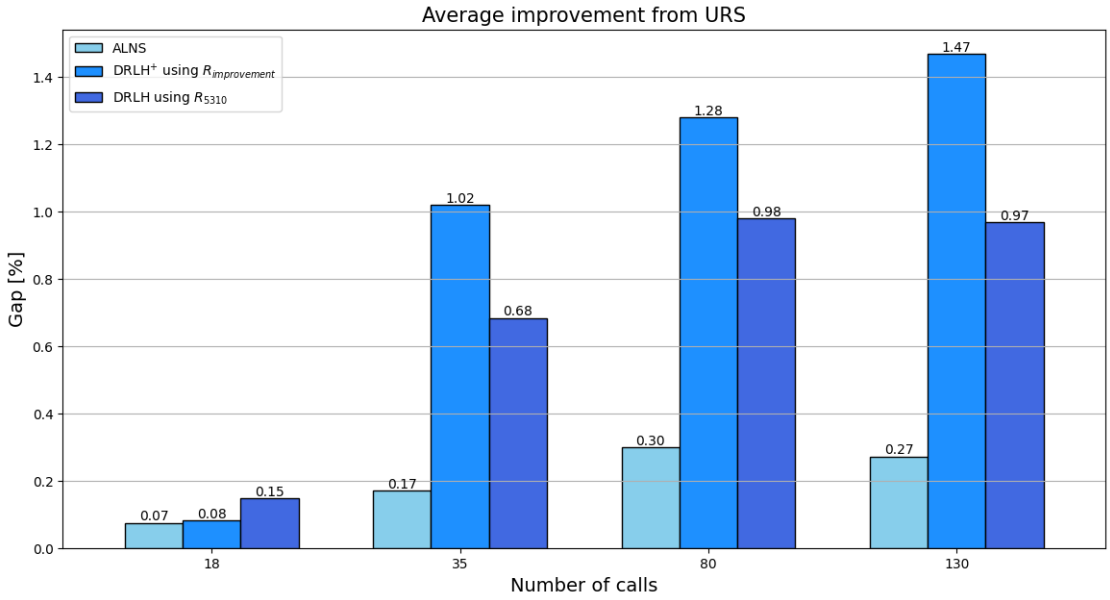


Figure 7.2: Performance comparison of $R_{improvement}$ and R_{5310}

The percentage improvement from the average cost found by URS, comparing the performance of the reward systems on each problem size. Average costs found after solving with 5 different seeds are compared to URS, and improvements are averaged over the 100 test instances.

In addition to the averaged improvements, table 7.1 reports how many instances each model found better solutions to than the other. The results are consistent with the averaged improvements, showing that DRLH$^+$ performs better on all sizes except the smallest. The largest difference is observed on 130 calls, with $R_{improvement}$ leading to better solutions on 79/100 instances.

7.3 Experiment on Search Progress

While the most important metric is the minimum cost found for an instance, it is also valuable to analyse how the results progress throughout the 1000 iterations of a search. This experiment considers the current best solution found at each iteration, and averages the results over 500 runs, consisting of 100 test instances solved with 5 different seeds. The two proposed models and all baselines are compared in figure 7.3, visualizing how the found minimum cost progresses throughout the search.

	Number of instances with better average solution		
Number of calls	$DRLH^+$ using $R_{improvement}$	$DRLH$ using R_{5310}	Total
18	24	76	100
35	65	35	100
80	64	36	100
130	79	21	100

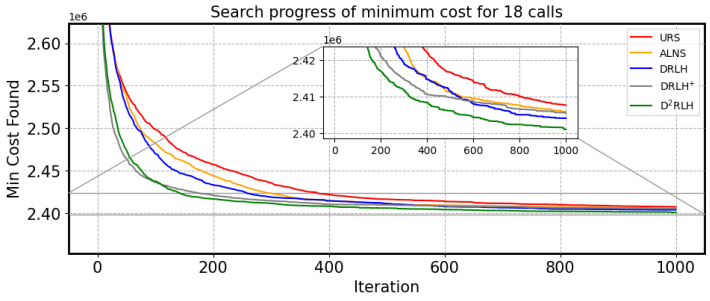
Table 7.1: Comparison of $R_{improvement}$ and R_{5310} on 100 instances

The share of the 100 test instances where the model achieved a better average solution than the other candidate, with results shown for each problem size. Costs are averaged after solving an instance with 5 different seeds.

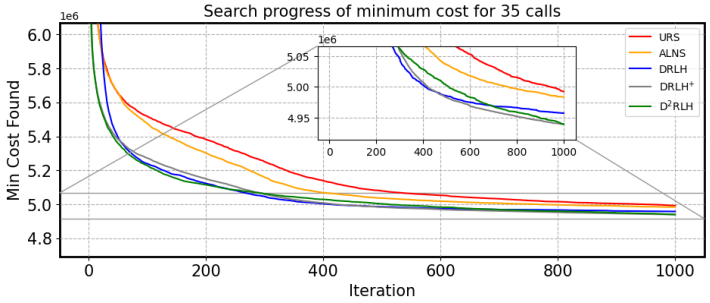
Models using DRL consistently outperform URS and ALNS on all problem sizes. In addition to achieving better final results, they much more efficiently reach good solutions. On all plots except 7.3a, the DRL models have already outperformed the final results of URS and ALNS by the midpoint of the search.

On all problem sizes except the smallest, figure 7.3 also displays a recurrent relation between DRLH and DRLH$^+$. Despite DRLH leading for the first half of the search, DRLH$^+$ eventually surpasses it and ultimately achieves a better result. The persistent improvement throughout the duration of the search is a promising trend, indicating that it avoids stagnation.

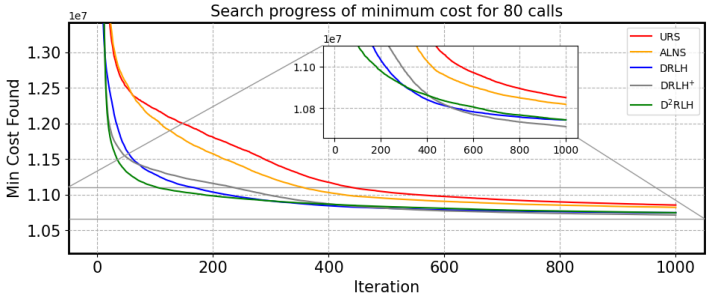
Figure 7.3d supports the explanation of the D^{2}RLH model's insufficiency suggested in section 7.1, based on the acceptance of worse solutions seen in figure A.1. The lack of diversification leads to rapidly improving solutions at the start, but also leads to earlier stagnation. For the smaller problem sizes, the moderate tempo ultimately results in better solutions. The reward function $R_{\eta \text{ steps ahead}}$ has a quite subjective definition of what the agent should achieve, by assigning both specific rewards to different outcomes and the number of steps it should expect to see results within. This might not be representative of the real objective, and the reward function could potentially benefit from a more direct relation to the objective value.



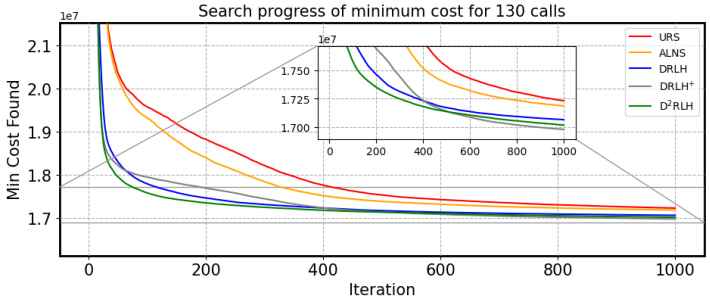
(a) 18 calls



(b) 35 calls



(c) 80 calls



(d) 130 calls

Figure 7.3: Averaged search progress of minimum cost found

The search progress of minimum cost found throughout the search, comparing the performance of all models on each problem size. Inline plots are shown with a restricted y-axis to differentiate the results for small values. Progress is averaged over 500 runs, consisting of 100 test instances solved with 5 different seeds.

7.3.1 Performance on Extended Searches

To test if the improvement demonstrated by $R_{improvement}$ can persist over time, an experiment is conducted where the test set for 130 call instances is solved for 5000 iterations instead of 1000. These extended searches are solved by the models trained with 1000 iterations, and averaged over 500 runs identically to the previous experiment on search progress. Figure 7.4 compares the resulting performance of DRLH$^+$ to DRLH.

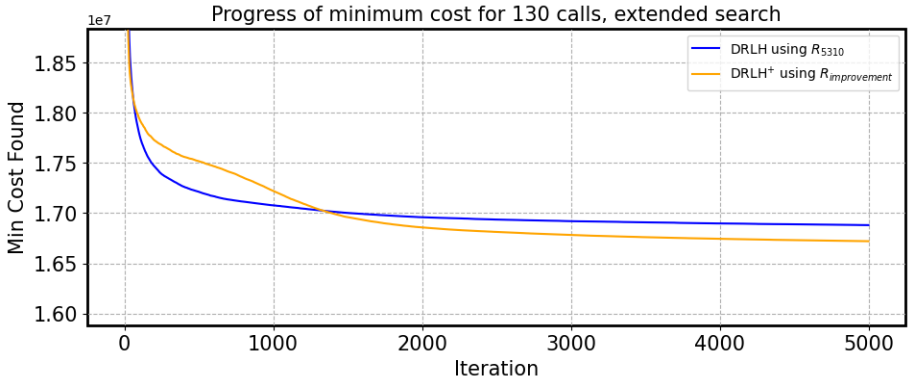


Figure 7.4: Progress comparison of $R_{improvement}$ and R_{5310} on extended runs

The search progress of minimum cost found throughout the extended search of 5000 iterations, comparing the performance of the reward systems on 130 call instances. Progress is averaged over 500 runs, consisting of 100 test instances solved with 5 different seeds.

DRLH$^+$ still displays a much better average performance than DRLH, with a clear separation in minimum cost by the end of the search. Notably, it takes more than 1000 iterations for DRLH$^+$ to surpass DRLH, despite achieving better results on a search length of 1000 iterations in the previous experiment. Since the *index* feature in the state representation shown in table 5.3 is normalized, it appears that the agent successfully adjusts to the length of the search. This results in a slow, yet consistent, pace of improvement regardless of search length.

Table 7.2 shows that the new reward function leads to a better average solution on 99 out of 100 test instances when solving with an extended search. It achieves even more dominant performance with more iterations available, almost exclusively outperforming the alternative.

	Number of instances with better average solution		
Number of calls	$DRLH^+$ using $R_{improvement}$	$DRLH$ using R_{5310}	Total
130	99	1	100

Table 7.2: Comparison of $R_{improvement}$ and R_{5310} on 100 instances, extended search

The share of the 100 test instances where the model achieved a better average solution than the other candidate, with results shown for 130 calls. Costs are averaged after solving an instance with 5 different seeds.

7.4 Experiment on Generalization with Benchmark Instances

The models trained on the problem sizes with nr_calls $\in \{18, 35, 80, 130\}$ are tested on the full set of problem sizes present in the benchmark instances. We have opted to solve each size with the first model that is trained on a size that is at least as large, resulting in the distribution in table 7.3. To shorten the names of the instance sets, full load cargo sizes and mixed cargo sizes are referred to as *fun* and *mun*, respectively.

Solved by model trained on	**Mun benchmark problem sizes**	**Fun benchmark problem sizes**
18 calls	7, 10, 15 and 18 calls	8, 11, 13, 16 and 17 calls
35 calls	22, 23, 30 and 35 calls	20, 25 and 35 calls
80 calls	60 and 80 calls	50 and 70 calls
130 calls	100 and 130 calls	90 and 100 calls

Table 7.3: Models used to solve benchmark problem sizes

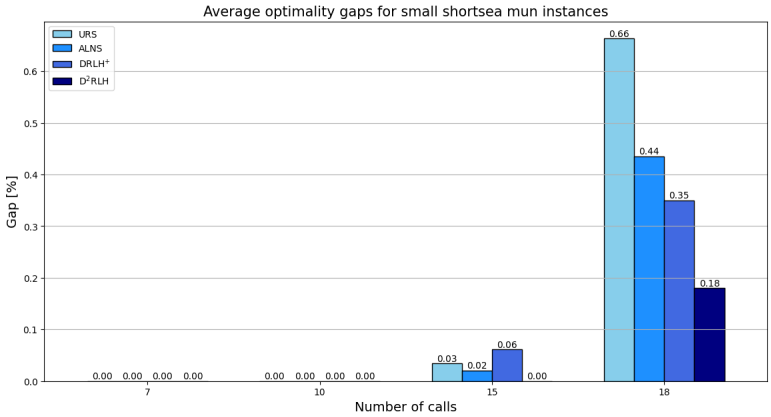
For each instance, a model's average cost found after solving with 5 different seeds is compared to the known optimal solution. The resulting optimality gap is averaged over the 5 instances with the same problem size within each group. The performances of the proposed models are compared to URS and ALNS for each benchmark set separately, with figures 7.5, 7.6, 7.7 and 7.8 displaying the results.

As the models are trained on *short sea + mun* instances, they are expected to exhibit better performance on this group of instances. Figure 7.5 shows that both intelligent models perform well on the small problem sizes, while being more inconsistent on the medium

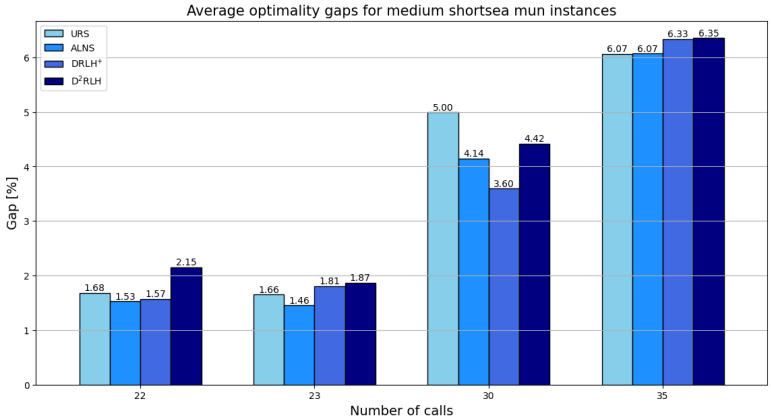
problem sizes. The models more consistently outperform the baselines as problem sizes increase, with DRLH$^+$ achieving the smallest optimality gaps on all of the large problem sizes.

The learnt behaviors of the models appear to generalize well to full load cargo sizes. In figure 7.6 which displays results for *short sea + fun* instances, there is a similar pattern across problem sizes to the first set. DRLH$^+$ and D^{2}RLH perform well on the large problem sizes, but are outperformed by ALNS on some of the middle sizes.

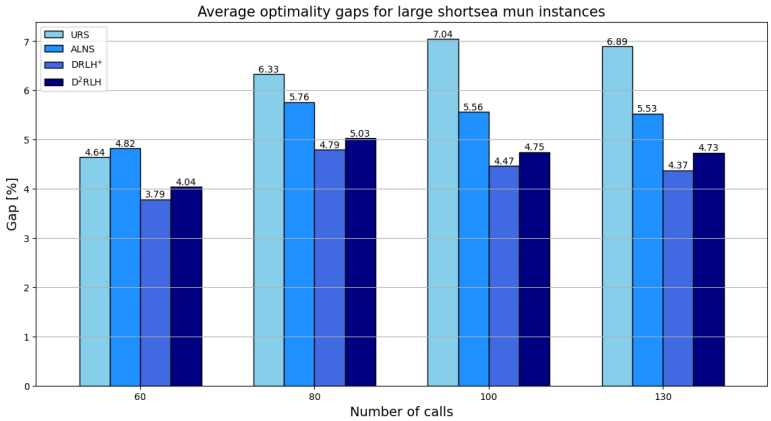
The results shown in figures 7.7 and 7.8 clearly demonstrate that the behavior learnt for short sea instances does not translate well to deep sea instances. In most cases, both models are even outperformed by the random selection of URS. Although short sea instances should theoretically be more difficult as locations are more densely clustered, a potential cause for the bad decisions could be a difference in dynamics between the instance types. The heuristics might have a different effect on the solutions when the dynamics differ, in which case there could be a lack of relevant training experience after only training on short sea instances.



(a) Short sea mun - Small instances



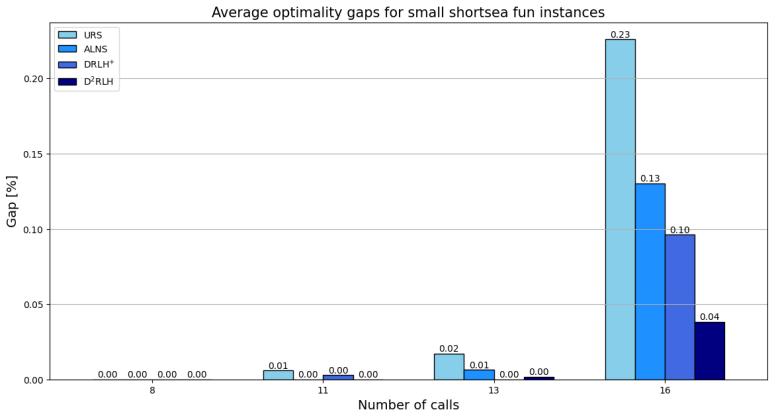
(b) Short sea mun - Medium instances



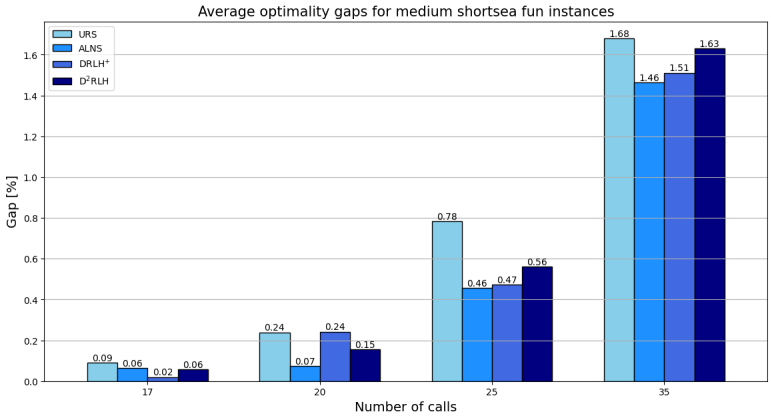
(c) Short sea mun - Large instances

Figure 7.5: Average optimality gaps for *Short sea + mixed cargo size* instances

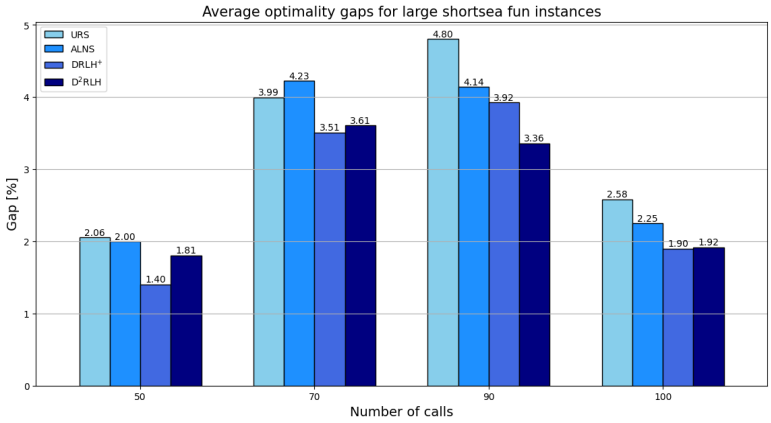
Comparison of average optimality gaps for 12 problem sizes of short sea instances with mixed cargo sizes. Optimality gaps are calculated for the average costs found after solving with 5 seeds, and gaps are averaged over the 5 instances with the same problem size.



(a) Short sea fun - Small instances



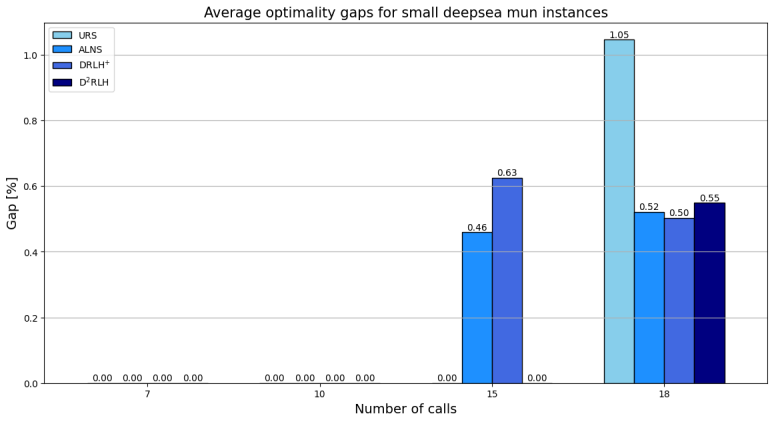
(b) Short sea fun - Medium instances



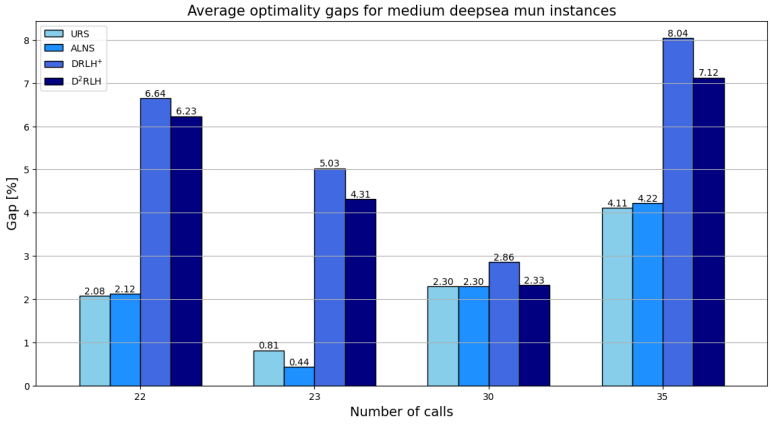
(c) Short sea fun - Large instances

Figure 7.6: Average optimality gaps for *Short sea + full load cargo size* instances

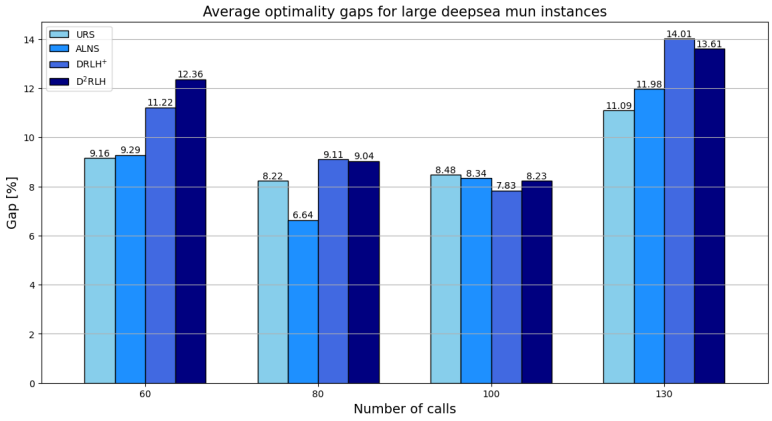
Comparison of average optimality gaps for 12 problem sizes of short sea instances with full load cargo sizes. Optimality gaps are calculated for the average costs found after solving with 5 seeds, and gaps are averaged over the 5 instances with the same problem size.



(a) Deep sea mun - Small instances



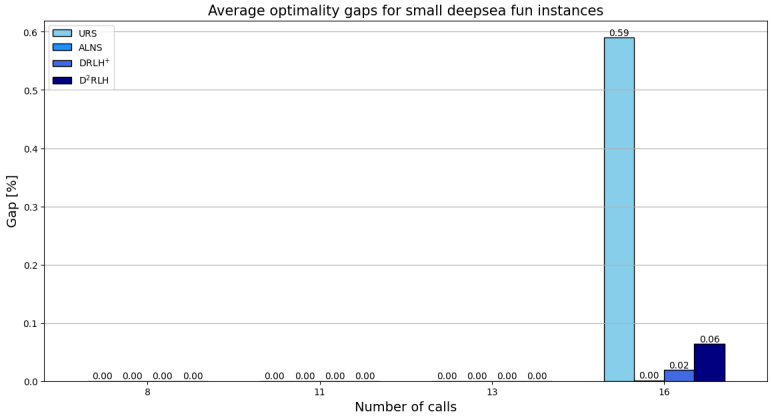
(b) Deep sea mun - Medium instances



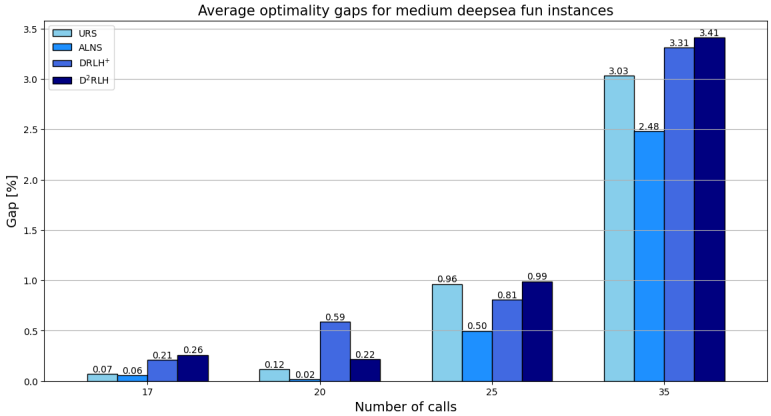
(c) Deep sea mun - Large instances

Figure 7.7: Average optimality gaps for *Deep sea + mixed cargo size* instances

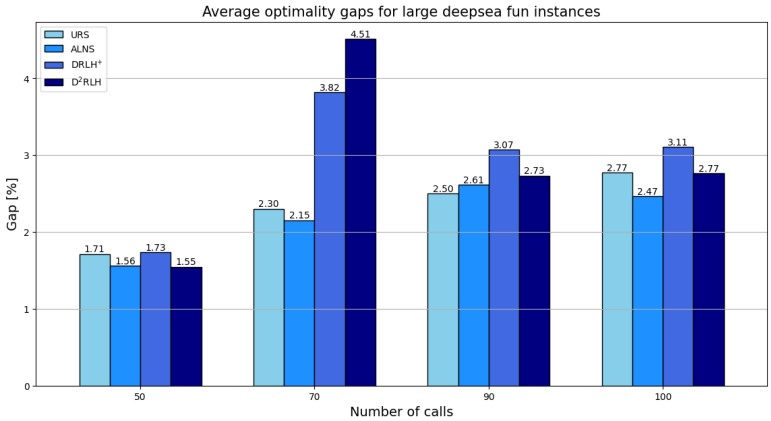
Comparison of average optimality gaps for 12 problem sizes of deep sea instances with mixed cargo sizes. Optimality gaps are calculated for the average costs found after solving with 5 seeds, and gaps are averaged over the 5 instances with the same problem size.



(a) Deep sea fun - Small instances



(b) Deep sea fun - Medium instances



(c) Deep sea fun - Large instances

Figure 7.8: Average optimality gaps for *Deep sea + full load cargo size* instances

Comparison of average optimality gaps for 12 problem sizes of deep sea instances with full load cargo sizes. Optimality gaps are calculated for the average costs found after solving with 5 seeds, and gaps are averaged over the 5 instances with the same problem size.

7.5 Experiment on Degree of Discounting

An additional experiment was conducted to analyse the performance of DRLH$^+$ with various degrees of discounting. The discount factor γ was set to multiple values, while keeping all other hyperparameters fixed to the values listed in section 6.2. The box plots in figure 7.9 present the results for each value, displaying the improvement compared to URS on the 100 test instances.

The results clearly suggest that a γ value of 0.99 is not suitable for the framework. The median improvement is small, and it is outperformed by URS on a large share of the instances. A potential explanation is the nature of a heuristic search. A single action can drastically change the solution, and the added randomness of the heuristics can cause large variations. A long trajectory of solutions will have a large variance, and attributing the full return to a single action could result in data that is too inconsistent to learn from.

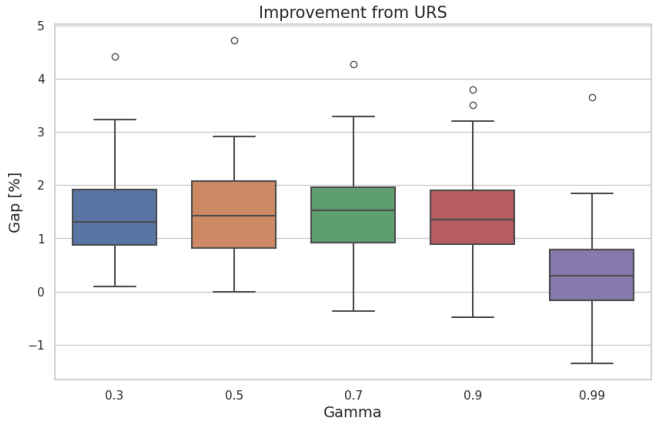


Figure 7.9: Performance of DRLH$^+$ for varying degrees of discounting

Box plot of percentage improvement from the average cost found by URS, comparing 5 values for the hyperparameter γ (gamma) on 130 call instances. Average costs found after solving with 5 different seeds are compared to URS, and improvements on 100 test instances are presented by the box plot.

Although it is clear that the presence of discounting is necessary, it is not as evident from figure 7.9 to what degree the returns should be discounted. The medians of the remaining values are somewhat similar, and there are substantial overlaps between the ranges of improvements. Based on this data, a value of 0.7 appears to be promising. It has the largest median improvement of the five tested values, and the small interquartile range suggests that it consistently achieves similar results.

Chapter 8

Conclusion

This thesis has presented DRLH$^+$ and D^{2}RLH, two hyperheuristic frameworks for combinatorial optimization problems using DRL. The design of the reward function for heuristic selection is emphasized, with both frameworks centered around a new reward function $R_{improvement}$. Tests performed on instances of PDPTW exhibit good performance compared to the baselines ALNS and DRLH. The frameworks show considerable improvements on large problem sizes, which is an advantage since these are the most difficult and time consuming to solve without heuristics.

Due to the new reward function, DRLH$^+$ maintains a steady progression of improvement throughout the full length of the search. An experiment on longer searches shows that it is able to adjust well to a larger number of iterations, and maintains consistent and gradual improvements that lead to good results by the end of the search. Although the discounting of the return is proven to be important, the degree of discounting appears to have little effect as long as it is included.

Adding the acceptance of D^{2}RLH appears to diminish the performance. While it still performs well compared to baselines, this is likely due to the heuristic selection. The reward function for acceptance leads to conservative strategies that rely too much on intensification. It could benefit from a similar type of reward function as $R_{improvement}$, that more directly defines the objective of the search.

A natural continuation of this work would be further development of the reward function $R_{improvement}$. Little parameter tuning was performed, and the balance between α and β has an

important impact on the balance between intensification and diversification. If the rewards for unseen solutions become too large, the agent will benefit from finding unique solutions regardless of their quality. DRLH$^+$ did not perform well on the smallest instances, and developing a version that performs well on all problem sizes would be a valuable contribution.

The benchmark tests indicated that good strategies for short sea instances do not extend to deep sea instances. To verify that the frameworks generalize well to other problem types, models could be trained and tested on deep sea instances. Additionally, assessing the performance on other combinatorial optimization problems is important to document it's suitability for a wider range of problems.

Future work should also attempt to improve the acceptance of D^{2}RLH by replacing $R_{\eta \text{ steps ahead}}$. A simpler reward function that has fewer manually set rewards could help mitigate reward hacking and better align the agent's objective with the real objective of the search. It is also encouraged to explore how the two agents are trained, with the possibility of formulating the framework as a multi-agent system.

Bibliography

[1] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete Problems in AI Safety, July 2016.
URL: http://arxiv.org/abs/1606.06565. arXiv:1606.06565 [cs].

[2] Edmund K. Burke, Graham Kendall, and Eric Soubeiga. A Tabu-Search Hyperheuristic for Timetabling and Rostering. *Journal of Heuristics*, 9(6):451–470, December 2003. ISSN 1572-9397. doi: 10.1023/B:HEUR.0000012446.94732.b6.
URL: https://doi.org/10.1023/B:HEUR.0000012446.94732.b6.

[3] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, December 2013. ISSN 1476-9360. doi: 10.1057/jors.2013.71.
URL: https://doi.org/10.1057/jors.2013.71.

[4] Yvan Dumas, Jacques Desrosiers, and François Soumis. The pickup and delivery problem with time windows. *European Journal of Operational Research*, 54(1):7–22, September 1991. ISSN 0377-2217. doi: 10.1016/0377-2217(91)90319-Q.
URL: https://www.sciencedirect.com/science/article/pii/037722179190319Q.

[5] Burak Eksioglu, Arif Volkan Vural, and Arnold Reisman. The vehicle routing problem: A taxonomic review. *Computers & Industrial Engineering*, 57(4):1472–1483, November 2009. ISSN 0360-8352. doi: 10.1016/j.cie.2009.05.009.
URL: https://www.sciencedirect.com/science/article/pii/S0360835209001405.

[6] Ahmad Hemmati and Lars M. Hvattum. Evaluating the importance of randomization in adaptive large neighborhood search. *International Transactions in Operational Research*, 24(5):929–942, 2017. ISSN 1475-3995. doi: 10.1111/itor.12273.

URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/itor.12273. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/itor.12273.

[7] Ahmad Hemmati, Lars M. Hvattum, Kjetil Fagerholt, and Inge Norstad. Benchmark Suite for Industrial and Tramp Ship Routing and Scheduling Problems. *INFOR: Information Systems and Operational Research*, 52(1):28–38, February 2014. ISSN 0315-5986. doi: 10.3138/infor.52.1.28.
URL: https://doi.org/10.3138/infor.52.1.28. Publisher: Taylor & Francis _eprint: https://doi.org/10.3138/infor.52.1.28.

[8] Gabriel Homsi, Rafael Martinelli, Thibaut Vidal, and Kjetil Fagerholt. Industrial and tramp ship routing problems: Closing the gap for real-scale instances. *European Journal of Operational Research*, 283(3):972–990, June 2020. ISSN 0377-2217. doi: 10.1016/j.ejor.2019.11.068.
URL: https://www.sciencedirect.com/science/article/pii/S0377221719309804.

[9] Eskil H. Isaksen. DRLMA: An Intelligent Move Acceptance for Combinatorial Optimization Problems based on Deep Reinforcement Learning. Master's thesis, The University of Bergen, July 2023.
URL: https://bora.uib.no/bora-xmlui/handle/11250/3084672. Accepted: 2023-08-17T23:39:52Z.

[10] Jakob V. Kallestad. Developing an Intelligent Hyperheuristic for Combinatorial Optimization Problems using Deep Reinforcement Learning. Master's thesis, The University of Bergen, September 2021.
URL: https://bora.uib.no/bora-xmlui/handle/11250/2827078. Accepted: 2021-11-02T00:58:39Z.

[11] Yuxi Li. Deep Reinforcement Learning: An Overview, November 2018.
URL: http://arxiv.org/abs/1701.07274. arXiv:1701.07274 [cs].

[12] Hao Lu, Xingwen Zhang, and Shuang Yang. A Learning-based Iterative Method for Solving Vehicle Routing Problems. In *International Conference on Learning Representations*, September 2019.
URL: https://openreview.net/forum?id=BJe1334YDH.

[13] Stefan Ropke and David Pisinger. An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. *Transportation Science*,

November 2006. doi: 10.1287/trsc.1050.0135.
URL: https://pubsonline.informs.org/doi/abs/10.1287/trsc.1050.0135. Publisher: INFORMS.

[14] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer Science & Business Media, February 2003. ISBN 978-3-540-44389-6. Google-Books-ID: mqGeSQ6dJycC.

[15] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust Region Policy Optimization. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 1889–1897. PMLR, June 2015.
URL: https://proceedings.mlr.press/v37/schulman15.html. ISSN: 1938-7228.

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017.
URL: http://arxiv.org/abs/1707.06347. arXiv:1707.06347 [cs].

[17] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation, October 2018.
URL: http://arxiv.org/abs/1506.02438. arXiv:1506.02438 [cs].

[18] Edward A. Silver. An overview of heuristic solution methods. *Journal of the Operational Research Society*, 55(9):936–956, September 2004. ISSN 1476-9360. doi: 10.1057/palgrave.jors.2601758.
URL: https://doi.org/10.1057/palgrave.jors.2601758.

[19] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, second edition: An Introduction*. MIT Press, November 2018. ISBN 978-0-262-35270-3. Google-Books-ID: uWV0DwAAQBAJ.

[20] Kenneth Sörensen and Fred W. Glover. Metaheuristics. In Saul I. Gass and Michael C. Fu, editors, *Encyclopedia of Operations Research and Management Science*, pages 960–970. Springer US, Boston, MA, 2013. ISBN 978-1-4419-1153-7. doi: 10.1007/978-1-4419-1153-7_1167.
URL: https://doi.org/10.1007/978-1-4419-1153-7_1167.

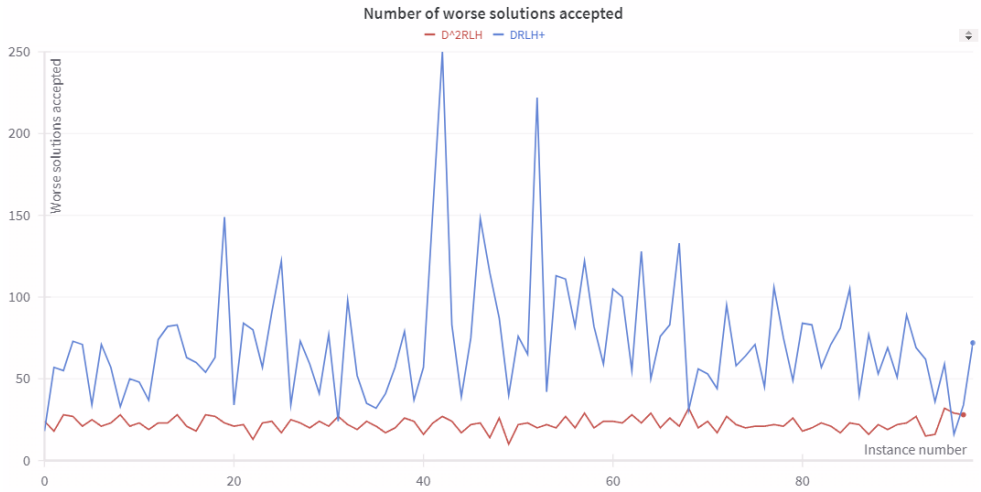
[21] United Nations Department of Economic and Social Affairs. Transforming our world: the 2030 Agenda for Sustainable Development.
URL: https://sdgs.un.org/2030agenda.

[22] Vard Group AS. Leading the way in the green maritime transition.
URL: https://www.vard.com/articles/the-ocean-charger-project-has-officially-started.

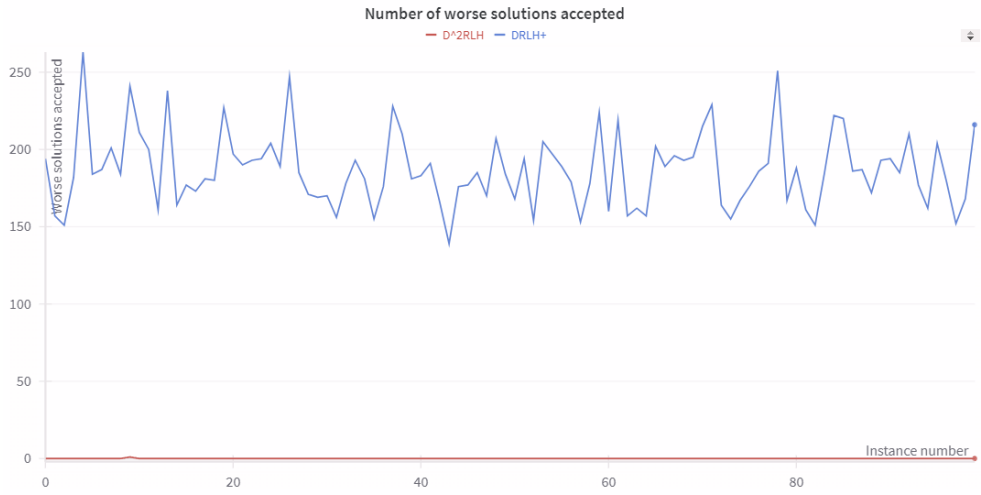
Appendix A

Acceptance Frequency for Experiment on Test Sets

Figure A.1 displays the number of times a solution that is worse than the current solution is accepted, for each of the 100 solved test instances. D^2RLH learns a more conservative approach, and rejects most worse solutions on small instances. On the problem size with 130 calls, the agent learns to never accept worse solutions. Based on the selected hyperparameters in table 6.7, this suggests that it leads too improvements too infrequently to benefit from accepting the solution. From the agent's perspective it is better to avoid the negative rewards than to rely on achieving positive rewards. This find indicates that the acceptance strategy is too shortsighted and needs to take longer trajectories into account.



(a) 18 calls



(b) 130 calls

Figure A.1: Number of worse solutions accepted per instance

Shows the number of worse solutions accepted by DRLH$^+$ using simulated annealing and by D^{2}RLH using DRL. Solutions that are accepted despite having larger cost than the current solution are counted, and the totals for each of the 100 test instances are displayed after solving with 1 seed.

Appendix B

Number of New Best Solutions for Reward System Comparison

Figure B.1 displays the number of times a new best solution is found, for each of the 100 solved test instances. Although DRLH is consistently outperformed by DRLH$^+$, it still improves upon the best solution substantially more often. Its reward function, R_{5310}, rewards the agent with a reward of 5 every time this occurs. These results support the theory of reward hacking taking place for DRLH, with the agent successfully maximizing the received rewards without achieving the true objective. While DRLH$^+$ improves less often, the average cost reduction for these improvements must be larger.

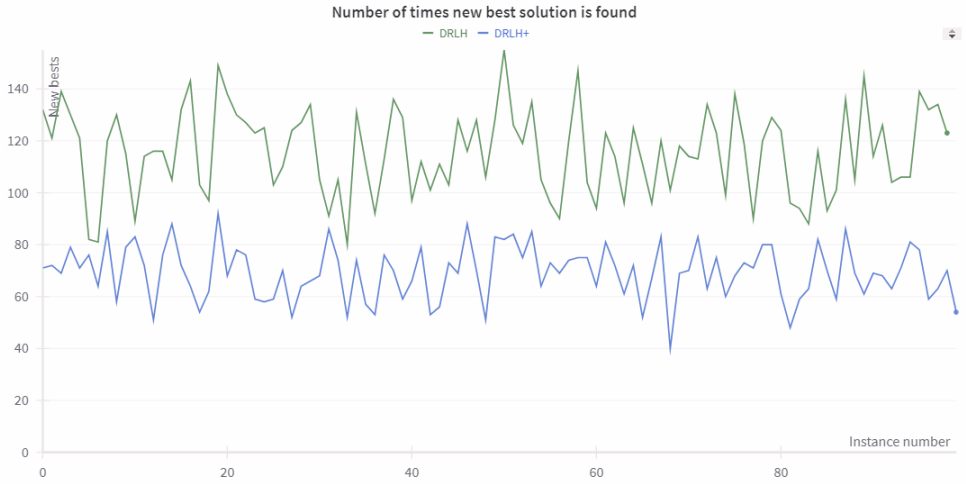


Figure B.1: Number of new best solutions per instance

Shows the number of new best solutions found by DRLH$^+$ using $R_{improvement}$ and by DRLH using R_{5310}
Every iteration where a solution that is better than the current best are counted,
and the totals for each of the 100 test instances are displayed after solving with 1 seed.