European Journal of Operational Research 309 (2023) 446–468

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Interfaces with Other Disciplines

A general deep reinforcement learning hyperheuristic framework for solving combinatorial optimization problems

Jakob Kallestad[a], Ramin Hasibi[a,*], Ahmad Hemmati[a], Kenneth Sörensen[b]

[a] Department of Informatics, University of Bergen, Norway
[b] Faculty of Business and Economics, ANT/OR - University of Antwerp Operations Research Group, Belgium

ARTICLE INFO

Article history:
Received 11 October 2021
Accepted 11 January 2023
Available online 16 January 2023

Keywords:
Heuristics
Hyperheuristic
Adaptive metaheuristic
Deep reinforcement learning
Combinatorial optimization

ABSTRACT

Many problem-specific heuristic frameworks have been developed to solve combinatorial optimization problems, but these frameworks do not generalize well to other problem domains. Metaheuristic frameworks aim to be more generalizable compared to traditional heuristics, however their performances suffer from poor selection of low-level heuristics (operators) during the search process. An example of heuristic selection in a metaheuristic framework is the adaptive layer of the popular framework of Adaptive Large Neighborhood Search (ALNS). Here, we propose a selection hyperheuristic framework that uses Deep Reinforcement Learning (Deep RL) as an alternative to the adaptive layer of ALNS. Unlike the adaptive layer which only considers heuristics' past performance for future selection, a Deep RL agent is able to take into account additional information from the search process, e.g., the difference in objective value between iterations, to make better decisions. This is due to the representation power of Deep Learning methods and the decision making capability of the Deep RL agent which can learn to adapt to different problems and instance characteristics. In this paper, by integrating the Deep RL agent into the ALNS framework, we introduce Deep Reinforcement Learning Hyperheuristic (DRLH), a general framework for solving a wide variety of combinatorial optimization problems and show that our framework is better at selecting low-level heuristics at each step of the search process compared to ALNS and a Uniform Random Selection (URS). Our experiments also show that while ALNS can not properly handle a large pool of heuristics, DRLH is not negatively affected by increasing the number of heuristics.

1. Introduction

A metaheuristic is an algorithmic framework that offers a coherent set of guidelines for the design of heuristic optimization methods. Classical frameworks such as Genetic Algorithm (GA), Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), and Simulated Annealing (SA) are examples of such frameworks (Dokeroglu, Sevinc, Kucukyilmaz, & Cosar, 2019). Moreover, there is a large body of literature that addresses solving combinatorial optimization problems using metaheuristics. Among these, Adaptive Large Neighbourhood Search (ALNS) (Ropke & Pisinger, 2006) is one of the most widely used metaheuristics. It is a general framework based on the principle of Large Neighbourhood Search (LNS) of Shaw (1998), where the objective value is iteratively improved by applying a set of "removal" and "insertion" operators on the solution. In ALNS, each of the removal and insertion operators have weights associated with them that determine the probabilities of selecting these during the search. These weights are continuously updated after a certain number of iterations (called a segment) based on their recent effect on improving the quality of the solution during the segment. The ALNS framework was early on an approach specific to routing problems. However, in recent years, there has been a growing number of studies that employ this approach to other problem types, e.g., scheduling problems (Laborie & Godard, 2007). Its high quality of performance at finding solutions has made it a go-to approach in many recent studies in combinatorial optimization problems (Aksen, Kaya, Sibel Salman, & Özge Tüncel, 2014; Chen, Demir, & Huang, 2021; Demir, Bektaş, & Laporte, 2012; Friedrich & Elbert, 2022; Grangier, Gendreau, Lehuédé, & Rousseau, 2016; Gullhav, Cordeau, Hvattum, & Nygreen, 2017; Li, Chen, & Prins, 2016). The ALNS framework has several advantages. For most optimization problems, a number of

* Corresponding author.
E-mail addresses: jakobkallestad@gmail.com (J. Kallestad), Ramin.Hasibi@uib.no (R. Hasibi), Ahmad.Hemmati@uib.no (A. Hemmati), kenneth.sorensen@uantwerpen.be (K. Sörensen).

https://doi.org/10.1016/j.ejor.2023.01.017
0377-2217/© 2023 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/)

well-performing heuristics are already known which can be used as the operators in the ALNS framework. Due to the large size and diversity of the neighborhoods, the ALNS algorithm will explore huge chunks of the solution space in a structured way. As a result, ALNS is very robust as it can adapt to different characteristics of the individual instances, and is able to avoid being trapped in local optima (Pisinger & Ropke, 2019). According to Turkeš, Sörensen, & Hvattum (2021), the adaptive layer of ALNS has only minor impact on the objective function value of the solutions in the studies that have employed this framework. Moreover, the information that the adaptive layer uses for selecting heuristics is limited to the past performance of each heuristic. This limited data can make the adaptive layer naïve in terms of decision making capability because it is not able to capture other (problem-independent) information about the current state of the search process, e.g., the difference in cost between past solutions, whether the current solution has been encountered before during the search, or the number of iterations since the solution was last changed, etc. We refer to the decision making capability of ALNS as performing on a “macro-level” in terms of adaptability, i.e., the weights of each heuristic is only updated at the end of each segment. This means that the heuristics selected within a segment are sampled according to the fixed probabilities of the segment. This limitation makes it impossible for ALNS to take advantage of any short-term dependencies that occur within a segment that could help aid the heuristic selection process.

Another area where ALNS struggles is when faced with a large number of heuristics to choose from. In order to find the best set of available heuristics for ALNS for a specific setting, initial experiments are often required to identify and remove inefficient heuristics, and this can be both time consuming and computationally expensive (Hemmati & Hvattum, 2017). Furthermore, some heuristics are known to perform very well for specific problem variations or specific conditions during the search, but they may have a poor average performance. In this case, it might be beneficial to remove these from the pool of heuristics available to ALNS in order to increase the average performance of ALNS, but this results in a less powerful pool of heuristics that is unable to perform as well during these specific problem variations and conditions.

To address the issues in ALNS, one can use Reinforcement Learning (RL). RL is a subset of machine learning concerned with “learning how to make decisions”—how to map situations to actions—so as to maximize a numerical reward signal. One of the main tasks in machine learning is to generalize a predictive model based on available training data to new unseen situations. An RL agent learns how to generalize a good policy through interaction with an environment which returns the reward in exchange for receiving an action from the agent. Therefore, through a trial-and-error search process, the agent is trained to achieve the maximum expected future reward at each step of decision making conditioned on the current situation (state). Thus, training an RL agent (to achieve the best possible results in similar situations), makes the agent aware of the dynamics of the environment as well as adaptable to similar environments with slightly different settings. One of the more recent approaches in RL is Deep RL which benefits from the powerful function approximation property of deep learning tools. In this approach, different functions that are used to train and make decisions in an RL agent are implemented using Artificial Neural Networks (ANNs). Different Deep RL algorithms dictate the training mechanism and interaction of the ANNs in the decision making process of the agent (Sutton & Barto, 2018). Therefore, integration of the Deep RL into the adaptive layer of the ALNS can make the resulting framework much smarter at making decisions at each iteration and improve the overall performance of the framework.

In this paper, we propose **Deep Reinforcement Learning Hyperheuristic (DRLH)**, a general approach to selection hyperheuristic framework (definition in Section 2) for solving combinatorial optimization problems. In DRLH, we replace the adaptive layer of ALNS with a Deep RL agent responsible for selecting heuristics at each iteration of the search. Our Deep RL agent is trained using Proximal Policy Optimization (PPO) method of Schulman, Wolski, Dhariwal, Radford, & Klimov (2017) which is a standard approach for stable training of the Deep RL agent in different environments. The proposed DRLH utilizes a search state consisting of a problem-independent feature set from the search process and is trained with a problem-independent reward function that encourages better solutions. This approach makes the framework easily applicable to many combinatorial optimization problems without any change in the method and given the proper training step for each problem separately. The training process of DRLH makes it adaptable to different problem conditions and settings, and ensures that DRLH is able to learn good strategies of heuristic selection prior to testing, while also being effective when encountering new search states. In contrast to the macro-level decision making of ALNS, the proposed DRLH makes decisions on a “micro-level”, meaning that only the current search state information affects the probabilities of choosing heuristics. This allows for the probabilities of selecting heuristics to change quickly from one iteration to the next, helping DRLH adapt to new information of the search as soon as it becomes available. The Deep RL agent in DRLH is able to effectively leverage this search state information at each step of the search process in order to make better decisions for selecting heuristics compared to ALNS.

To evaluate the performance and generalizability of DRLH, we choose four different combinatorial optimization problems to benchmark against different baselines in terms of best objective found and the speed of convergence as well as the time it takes to solve each problem. These problems include the Capacitated Vehicle Routing Problem (CVRP), the Parallel Job Scheduling Problem (PJSP), the Pickup and Delivery Problem (PDP), and the Pickup and Delivery Problem with Time Windows (PDPTW). These problems are commonly used for evaluation in the literature and are diverse in terms of difficulty to find good and feasible solutions. They additionally correspond to a broad scope of real world applications. For each problem, we create separate training and test datasets. In our experiments, we compare the performance of DRLH on different problem sizes and over an increasing number of iterations of the search and demonstrate how the heuristic selection strategy of DRLH differs from other baselines throughout the search process.

Our experiments show the superiority of DRLH compared to the popular method of ALNS in terms of performance quality. For each of the problem sets, DRLH is able to consistently outperform other baselines when it comes to best objective value specifically in larger instances sizes. Additionally, DRLH does not add any overhead to the instance solve time and the performance gain is a result of the decision making capability of the Deep RL agent used. Further experiments also validate that unlike other algorithms, the performance of DRLH is not negatively affected by increasing the number of available heuristics to choose from. In contrast to this, ALNS struggles when handling a large number of heuristics to choose from. This advantage of our framework makes the development process for DRLH very simple as DRLH seems to be able to automatically discover the effectiveness of different heuristics during the training phase without the need for initial experiments in order to manually reduce the set of heuristics.

The remainder of this paper is organized as follows: In Section 2, related previous work in hyperheuristics and Deep RL is presented. In Section 3, we propose the overall algorithm of DRLH as well as the choice of heuristics and parameters. The descriptions of the four combinatorial optimization problems used for benchmarking purposes are illustrated in Section 4. The experimental setup and the results of our evaluation are presented in Sections 5 and 6, respectively.

2. Related work

In this section, we first define the term “Hyperheuristic” and review some of the traditional work that fall into this category and point out their limitations. We also mention some of the methods that employ Deep RL for solving combinatorial problems and their shortcomings. In the end, we explain how we combine the best of two domains (Hyperheuristic and Deep RL) to take advantage of both their methodologies.

The term *hyperheuristic* was first used in the context of combinatorial optimization by Cowling, Kendall, & Soubeiga (2001) and described as *heuristics to choose heuristics*. Burke et al. (2010) later extended the definition of hyperheuristic to “a search method or learning mechanism for selecting or generating heuristics to solve computational search problems”. The most common classification of hyperheuristics makes the distinction between *selection hyperheuristics* and *generation hyperheuristic*. Selection hyperheuristics are concerned with creating a selection mechanism for heuristics at each step of the search, while generation hyperheuristics are concerned with generating new heuristics using basic components from already existing heuristic methods. This paper will focus on selection hyperheuristics methods.

Although it is possible to create highly effective problem-specific and heuristic-specific methods for heuristic selection, these methods do not always generalize well to other problem domains and different sets of heuristics. A primary motivation of hyperheuristic research is therefore the development of general-purpose, problem-independent methods that can deliver good quality solutions for many combinatorial optimization problems without having to make significant modifications to the methods. Thus, advancements done in hyperheuristic research aims to be easily applicable by experts and non-experts alike, to various problems and heuristics sets without requiring extra effort such as domain knowledge about the specific problem to be solved.

A classic example of using RL in hyperhueristics is the work of Özcan, Misir, Ochoa, & Burke (2010) in which they propose a framework that uses a traditional RL method for solving examination timetabling. Performance is compared against a simple random hyperheuristic and some previous work, and results show that using RL obtains better results than simply selecting heuristics at random. The RL used here learns during the search process by adjusting the probabilities of choosing heuristics based on their recent performance during the search. This type of RL framework shares many similarities with the ALNS framework, and therefore suffers from the same limitations as those mentioned for ALNS.

Apart from RL, supervised learning, which is another machine learning technique, has also been utilized in hyperheuristic frameworks to improve the performance. A hyperheuristic method for the Vehicle Routing Problem named *Apprentice Learning-based Hyper-heuristic (ALHH)* was proposed by Asta & Özcan (2014) in which an apprentice agent seeks to imitate the behavior of an expert agent through supervised learning. The training of the ALHH works by running the expert on a number of training instances and recording the selected actions of the expert together with a *search state* that consists of the previous action used and the change in objective function value for the past n steps. These recordings of search state and action pairs build up a training dataset in which a decision tree classifier is used in order to predict the action choice of the expert. This makes up a supervised classification problem in which the final accuracy of the model is reported to be around 65%. In the end ALHH’s performance is compared against the expert and is reported to perform very similarly to the expert, and even slightly outperforming the expert for some instances.

Tyasnurita, Özcan, Shahriar, & John (2015) further improved upon the apprentice learning approach by replacing the decision tree classifier with a multilayer perceptron (MLP) neural network, and named their approach MLP-ALHH. This change increased the representational power of the search state and resulted in a better performance that is reported to even outperform the expert. A limitation of ALHH and MLP-ALHH is their use of the supervised learning framework which makes performance of these approaches bounded by the expert algorithm’s performance. A consequence of this is that the feedback used to train the predictive models of ALHH and MLP-ALHH is binary, i.e. it either matches that of the expert or not, leaving no room for alternative strategies that might perform even better than the expert. In contrast, DRLH uses a Deep RL framework that neither requires, nor is bounded by an expert agent and therefore has more potential to outperform existing methods by coming up with new ways of selecting heuristics. The feedback used to train DRLH depends on the effect of the action on the solutions, and the amount received varies depending on several factors. Additionally, DRLH takes future iterations of the search into account, while ALHH and MLP-ALHH only consider the immediate effect of the action on the solution. Because of this, diversifying behavior is encouraged in DRLH when it gets stuck, as it will help improve the solution in future iterations. Another difference of DRLH compared to ALHH and MLP-ALHH is that the features of the search state used by DRLH contain more information compared to the search state of the other two methods which ultimately makes the agent more aware of the search state and thus capable of making effective decisions.

In addition to hyperheuristic approaches there have also recently been many attempts at solving popular routing problems using Deep RL by the machine learning community. A big limitation of these works is that they all rely on problem-dependent information, and are usually designed to solve a single problem or a small selection of related problems, often requiring significant changes to the approach in order to make them work for several problems. In first versions of these studies, Deep RL is used as a constructive heuristic approach for solving the vehicle routing problem in which the agent, representing the vehicle, selects the next node to visit at each time step (Kool, van Hoof, & Welling, 2019; Nazari, Oroojlooy, Snyder, & Takac, 2018). Although this is very effective when compared to simple construction heuristics for solving routing problems, it lacks the quality of solutions provided by iterative metaheuristic approaches as well as being unable to find feasible solutions in the case of more difficult routing problems that involve more advanced constraints such as pickup and delivery problem with time windows.

Another approach that leverages Deep RL for solving combinatorial optimizations is to take advantage of the decision making ability of the agent in generating or selecting low-level heuristics to be applied on the solution. Hottung & Tierney (2019) have used a Deep RL agent to generate a heuristic for rebuilding partially destroyed routes in the CVRP using a large neighbourhood search framework. This method is an example of heuristic generation and is specifically designed to solve the CVRP. Thus, it can not easily be generalized to other problem domains. In Chen & Tian (2019), a framework is presented for using two Deep RL agents for finding a node in the solution and the best heuristic to apply on that node at each step. Although the authors claim that this method is generalizable to three different combinatorial optimization problems, the details in representation of the problem and type of ANNs used for the agents from one problem to another change a lot depending on the nature of the problem. Additionally, one would have to come up with new inputs and representation when applying this method to other optimization problems that are not discussed in the study which reduces the generalizability of the framework. Lu, Zhang, & Yang (2020) suggested the use of a Deep RL agent for choosing low-level heuristic at each step for the CVRP. This work also suffers from the generalizability to other types of optimization problems due to the elements of the Deep RL agent that are specific to the CVRP problem. Additionally, in this approach the training process of the agent is designed in such a way that the agent is only focused on intensification rather than diversification. Thus, the diversification in their framework is done by a rule-based escape approach rather than giving the RL agent freedom to find the balance between diversification and intensification, which could lead to better results.

To the best of our knowledge previous work on this topic either suffer from a lack of generalizability in approach when it comes to other problems in the domain or they do not take advantage of the learning mechanism and representation power of Deep RL. In this work we seek to address these issues by introducing DRLH.

3. DRLH

In this section, we present the DRLH, a hyperheuristic framework to solve combinatorial optimization problems.

Our proposed hyperheuristic framework uses an RL agent for the selection of heuristics. This process improves on the ALNS framework of Ropke & Pisinger (2006) by leveraging the RL agent's decision making capability in choosing the next heuristic to apply on the solution in each iteration. The pseudocode of DRLH is illustrated in Algorithm 1.

Algorithm 1: DRLH.

```
Function Deep Reinforcement Learning Hyperheuristic
    Generate an initial solution $x$ with objective function of $f(x)$ (see section 3.5)
    $H$=Generate_heuristics() (see section 3.1)
    $x_{best} = x, f(x_{best}) = f(x)$
    Repeat
        $x' = x$
        choose $h \in H$ based on policy $\pi(h|s,\theta)$ (see section 3.3)
        Apply heuristic $h$ to $x'$
        if $f(x') < f(x_{best})$ then
            $x_{best} = x$
        end
        if $accept(x', x)$ (see section 3.3), then
            $x = x'$
        end
    Until stop-criterion met (see section 3.4)
    return $x_{best}$
```

3.1. Generating heuristics

The heuristic generation process follows the steps in Algorithm 2. The set H consists of all possible heuristics that can be applied on the solution x at each iteration. The general method for obtaining these heuristics is to combine a removal and an insertion operator. Furthermore, additional heuristics can also be placed in H that do not share the characteristic of being a combination of removal and insertion operators. In the following, we present one example set of H for the problem types considered for this paper.

Algorithm 2: Generation of the set of heuristics H.

```
Function Generate_heuristics
    H={};
    foreach removal operator $r \in \mathcal{R}$ do
        foreach insertion operator $j \in \mathcal{I}$ do
            Create a heuristic $h$ by combining $r$ and $j$;
            $H = H \cup h$;
        end
    end
    foreach additional heuristic $c \in \mathcal{C}$ do
        $H = H \cup c$;
    end
    return $H$
```

3.2. Sample set of heuristics

Each heuristic $h \in H$ is a combination of a removal and an insertion operator presented in Tables 1 and 2. Furthermore, one additional intensifying heuristic is also added to H. In each iteration, a heuristic $h \in H$ is applied on the incumbent solution x with cost of $f(x)$ and generates a new solution x' with cost of $f(x')$. For our sample set of heuristics, H has the size of $|H| = 29$ (7 removals $\times$ 4 insertions + 1 additional).

3.2.1. Removal operators $\mathcal{R}$

The set of all removal operators $\mathcal{R}$ are provided in Table 1. Seven removal operators are implemented, five of which are focused on inducing diversification through a high degree of randomness denoted by *Random* in their name. For intensification purposes, we define the operator "*Remove_largest_*$\mathcal{D}$" which uses the metric *Deviation* $\mathcal{D}$. We define the deviation D_i as the difference in cost with and without *element*$_i$ in the solution, and thus "*Remove_largest_*$\mathcal{D}$" removes the elements with the largest D_i. Finally, "*Remove_*τ" operator selects a number of consecutive elements in the solution and removes them.

3.2.2. Insertion operators $\mathcal{I}$

Table 2 lists the set of insertion operators $\mathcal{I}$ used. A total of 4 insertion operators are utilized to place the removed elements in a suitable position in solution x'. Operator "*Insert_greedy*" places each removed element in the position which obtains the minimum total cost of the new solution $f(x')$. Operator "*Insert_beam_search*" performs beam search with a search width of 10 for inserting each removed element. Beam search keeps track of the 10 best combinations of positions after inserting each removed element in the solution and inserts the elements in the best combination of positions that obtain the minimum $f(x')$ in the search space. The "*Insert_by_variance*" operator calculates the variance of the ten best insertion positions for each of the removed elements. Then the elements are ordered from high to low variance and inserted back into the solution with the "*Insert_greedy*" operator. Finally, operator "*Insert_first*" places each removed element randomly in the first feasible position found in the new solution.

3.2.3. Additional heuristic $\mathcal{C}$

Unlike in ALNS where only removal and insertion operators are used, our framework can also make use of standalone heuristics that share neither of the these types of characteristics. An example of one such additional heuristic, "*Find_single_best*", is responsible for generating the best possible new solution from the incumbent by changing one element. This heuristic calculates the cost of removing each element and re-inserting it with "*Insert_greedy*", and applies this procedure on the solution x for the element that

Table 1
List of all removal operators.

Name	Description
Random_remove_XS	Removes between 2–5 elements chosen randomly
Random_remove_S	Removes between 5–10 elements chosen randomly
Random_remove_M	Removes between 10–20 elements chosen randomly
Random_remove_L	Removes between 20–30 elements chosen randomly
Random_remove_XL	Removes between 30–40 elements chosen randomly
Remove_largest_$\mathcal{D}$	Removes 2–5 elements with the largest $\mathcal{D}_i$
Remove_τ	Removes a random segment of 2–5 consecutive elements in the solution

Table 2
List of all insertion operators.

Name	Description
Insert_greedy	Inserts each element in the best possible position
Insert_beam_search	Inserts each element in the best position using beam search
Insert_by_variance	Sorts the insertion order based on variance and inserts each element in the best possible position
Insert_first	Inserts each element randomly in the first feasible position

achieves the minimum cost $f(x')$. "*Find_single_best*" is the only additional heuristic that is used in the proposed sample set of heuristics, H.

3.3. Acceptance criteria and stopping condition

We use the acceptance criterion $accept(x', x)$ used in simulated annealing (Kirkpatrick, Gelatt, & Vecchi, 1983). This acceptance criterion depends on the difference in objective value between the incumbent x and the new solution x' denoted as $\Delta E = f(x') - f(x)$ together with a temperature parameter T that is gradually decreasing throughout the search. A new solution is always accepted if it has a lower cost than the incumbent, $\Delta E < 0$. In addition, worse solutions are accepted with probability $e^{-|\Delta E|/T}$.

To determine the initial temperature T_0 we accept all solutions for the first 100 iterations of the search and keep track of all the non-improving steps, $\Delta E > 0$. Then, we calculate the average of these positive deltas $\overline{\Delta E}$ in order to get:

$$T_0 = \frac{\overline{\Delta E}}{\ln 0.8} \tag{1}$$

To decrease the temperature we use the cooling schedule of Crama & Schyns (2003), and the search terminates after a certain number of iterations has been reached.

3.4. Deep RL agent for selection of h

In a typical RL setting, an agent is trained to optimize a policy π for choosing an action through interaction with an environment. At each time step (iteration) t, the agent chooses an action A_t and receives a scalar reward R_t from the environment indicating how good the action was. State S_t is defined as the information received at each time step from the environment based on the agent's choice of action A_t from a set of possible actions. Thus, a stochastic policy π for the agent is defined as

$$\pi(a|s) = Pr\{A_t = a | S_t = s\}. \tag{2}$$

One such type of policy is the parameterized stochastic policy function in which the probability of action selection is also conditioned on a set of parameters $\theta \in \mathcal{R}^d$. As a result, Eq. (2) is redefined as

$$\pi(a|s, \theta) = Pr\{A_t = a | S_t = s, \theta_t = \theta\}. \tag{3}$$

in which θ_t represents the parameters at time step t (Sutton & Barto, 2018). In our setting, the policy π is a MultiLayer perceptron (MLP), which is a class of non-linear function approximation (Goodfellow, Bengio, & Courville, 2016). In this scenario, the aim is to obtain the optimal policy π^* by tuning θ which represents the weights of the MLP network.

The training process for an RL agent is illustrated in Algorithm 3. For training the weights of the MLP, we follow the

Algorithm 3: Training the Deep RL agent.

```
Result: $\pi^*$ optimal policy
Start with random setting of $\theta$ for a random policy $\pi$;
for $e \leftarrow 1$ to episodes do
    Receive initial state $S_1$;
    for $t \leftarrow 1$ to steps do
        choose and perform action $a \in A_t$ according to
        $\pi(a|s, \theta)$;
        Receive $R_t = v$ and $s \in S_{t+1}$ from the environment
    end
    Optimize the policy parameters $\theta$ according to PPO
    (Schulman et~al., 2017).
end
```

policy gradient method of PPO introduced in Schulman et al. (2017). In order to generalize to different variations of an optimization problem, the training process is done for a number of problem instances (episodes) with each instance corresponding to a different set of attributes of the problem. Each instance is optimized for a certain number of iterations (time steps) and at the end of each episode the policy parameters θ are updated until we obtain the optimal policy. Once the training process is complete, the optimal policy π^* is used to solve unseen instances in the test sets.

As mentioned above, three main properties of the RL agent which are used to obtain the optimal policy π^* for solving the intended problem are the *state representation*, the *action space*, and the *reward function*. These parameters dictate the training process and decision making capability of the agent and are therefore essential for obtaining good solutions to optimization problems. Moreover, in our proposed approach, these properties are set to be independent of the type of problem which helps this approach generalize to many types of combinatorial optimization problems. The state representation contains the information about the current solution and the overall search state, and is shown to the agent at each step in order to guide the agent in the action selection process. The action space consists of a set of interchangeable heuristics that can be selected at each time step by the agent. Finally, the reward function guides the learning of the agent during

Table 3
A list of all features used for the state representation.

Name	Description
reduced_cost	The difference in cost between the previous & the current solutions
*cost_from_*min	The difference in cost between the current & the best found solution
cost	The cost of the current solution
min_cost	The cost of the best found solution
temp	The current temperature
cs	The cooling schedule (α)
no_improvement	The number of iterations since the last improvement
index_step	The iteration number
was_changed	1 if the solution was changed from the previous, 0 otherwise.
unseen	1 if the solution has not previously been encountered in the search, 0 otherwise.
last_action_sign	1 if the previous step resulted in a better solution, 0 otherwise.
last_action	The action in previous iteration encoded in 1-hot.

training and should be designed in a way that helps the agent optimize the objective of the problem. In the following, we explain the choice for each of these properties.

3.4.1. State representation

The state consists of a set of useful features for guiding the agent to select the best action/heuristic at each iteration in the search. We have prioritized general state features that are independent of the specifics of the problem being solved. In other words, the state representation is easily applicable to many optimization problems of different domains. Table 3 lists all the state features used by the agent.

The state features *cost* and *min_cost* together with *index_step* allow the agent to know approximately how well it is doing during the search. This becomes apparent if *cost* and *min_cost* are higher than their average values during training with respect to *index_step*. These state features primarily help at a macro-level by making the agent stick to a high-level strategy of heuristic selection throughout the search. *cost_from_*min, *temp*, *cs* and *no_improvement* inform the agent about how likely a new solution is to be accepted. These state features help the agent know how much intensification/diversification is appropriate at that step. For instance if it should try to escape a local optima or if it should focus on intensification. The last five state features; *reduced_cost*, *was_changed*, *unseen*, *last_action_sign* and *last_action* inform the agent about the immediate changes from the previous solution to the current solution. In particular, *reduced_cost* shows the difference in cost between the previous and current solution. *was_changed* indicates if the solution was changed from the previous step to the current step. *unseen* indicates whether the current solution was encountered before during the search. Finally, *last_action_sign* indicates if the solution improved or worsened from the previous step, and *last_action* indicates the action that was used in the previous step. Together these five features give information about what action the agent selected in the previous step and the result of that action. This helps the agent make decisions at a micro-level and is particularly useful as the agent can avoid selecting deterministic or semi-deterministic heuristics such as *Remove_largest_$\mathcal{D}$*, *Insert_by_variance* and *Find_single_best* twice in a row if the first time did not lead to any improvement, because then it is less likely, if at all, to work the second time on the same solution. This is particularly important for *Find_single_best* which is a fully deterministic heuristic and produces the same result if applied for two consecutive iterations.

3.4.2. Action

The actions in our setting for the agent are the same as the set of heuristics H, i.e, $A_t = H$. At each iteration of the DRLH (c.f., Algorithm 1), a heuristic h is selected and applied on the solution by the agent. Therefore the policy function π in Eq. (3) is redefined as

$$\pi(h|s,\theta) = Pr\{A_t = h|S_t = s, \theta_t = \theta\}. \tag{4}$$

3.4.3. Reward function

A good reward function needs to balance the need for gradual and incremental rewards while also preventing the agent from exploiting the reward function without actually optimizing the intended objective (also known as reward hacking Amodei et al., 2016). For our framework, we propose a reward functions that has the above property. We refer to this as R_t^{5310}, the formula for which is

$$R_t^{5310} = \begin{cases} 5, & \text{if } f(x') < f(x_{best}) \\ 3, & \text{if } f(x') < f(x) \\ 1, & \text{if } accept(x', x) \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

R_t^{5310} is inspired from the scoring mechanism that is applied in the ALNS framework for measuring the performance of each heuristic in a segment. This reward function encourages the agent to find better solutions than the current one as this gives a high reward. In addition it also gives a small reward if it finds a slightly worse solution that manages to get accepted by the acceptance criterion. This property of the function in turn motivates the agent to use diversifying operators when it is no longer able to improve upon the current solution. Moreover, other reward functions were considered for the framework which take the step-wise improvement of the solution as well as the amount of improvement into account. Further experiments on these reward functions demonstrate that the R_t^{5310} proved to be more stable and faster to train compared to the others (results in Appendix A). Furthermore, given the fact that R_t^{5310} comes from the original scoring function of ALNS in Ropke & Pisinger (2006), we use the same function for our Deep RL agent and ALNS for an equal comparison.

3.5. Solution representation and initial solution

For all the problems described in Section 4, the solution is represented as a permutation of orders/calls/jobs on each of the available vehicles/machines. Additionally, for the PDP and PDPTW, each call should be in the solution twice, one time for each of the pickup and the delivery elements respectively, and no call can be present in multiple vehicles, as the same vehicle has to both pick up and deliver the call.

The initial solutions for all of the problems are created by inserting all the orders/calls/jobs into the vehicles/machines using the *insert_greedy* operator from Table 2. For each of the problems and each test instance, DRLH, ALNS and URS start with the same initial solution for a fair comparison.

4. Problem sets

We consider four sets of combinatorial optimization problems as examples of problems that can be solved using DRLH. These problems are the Capacitated Vehicle Routing Problem (CVRP), Parallel Job Scheduling Problem (PJSP), Pickup and Delivery Problem (PDP) and Pickup and Delivery Problem with Time Windows (PDPTW).

4.1. CVRP

The Capacitated Vehicle Routing Problem is one of the most studied routing problems in the literature. It consists of a set of N orders that needs to be served by any of the M number of vehicles. Additionally, there is a depot in which the vehicles travel from and return to when serving the orders. Following the previous work, the number of vehicles in this particular problem is not fixed, but is naturally limited to $M = \{1, \ldots, N\}$. Meaning that the maximum number of vehicles that can be utilized is N and the minimum number is 1. Usually the number of vehicles used will fall somewhere in between depending on which number results in the best solution. Each order has a weight W_i associated to it, and the vehicles have a maximum capacity. The sequence of orders that a vehicle visits after leaving the depot before returning to the depot is referred to as a *tour*. There needs to be a minimum of one tour and a maximum of N tours. The combined weight of the orders in a tour can not exceed the maximum capacity of the vehicle, and so several tours are often needed in order to solve the CVRP problem. The objective of this problem is to create a set of tours that minimize the total distance travelled by all the vehicles that are serving at least one order.

4.2. PJSP

In the Parallel Job Scheduling Problem, we are given N jobs and M machines. Each of the machines operate with a different processing speed, and so the time required to complete job i on machine m is $T_{i,m}$. Each job has a *due time* associated with it, and if a job is finished after its due time, a delay is calculated for that job. The delay for job i is the difference in time between the due time and the actual finishing time of job i, and can never be lower than 0. The objective of the problem is to find a sequence of jobs to complete on each of the machines in order to minimize the total delay of all the jobs.

4.3. PDP

In Pickup and Delivery Problem we are given N calls and a single vehicle with a maximum capacity. Each call has a weight, a pickup location, and a delivery location, and is served when the order is transported by the vehicle from the pickup to the delivery location. The objective of the problem is to minimize the traveling distance of the vehicle while serving all the calls and not carrying more than the maximum capacity at any point.

4.4. PDPTW

In pickup and delivery problem, we are given a set of calls. A call consists of an origin and a destination and an amount of goods that should be transported. A heterogeneous fleet of vehicles are serving the calls, picking up goods at their origins and delivering them to their destinations. Time windows are assigned to each call at origins and destinations. Pickups and deliveries must be within the associated time windows. In the event of early arrival of a vehicle to a node before the start of the time window, the mentioned vehicle must wait until the beginning of the time window before being able to perform the pick up or delivery. A vehicle is never allowed to arrive at a node after the end of the time window. Additionally, a service time is considered for each time a call gets picked up or delivered, i.e., the time it takes a vehicle to load or deliver the goods at each node. For each call, a set of feasible vehicles is determined. Each vehicle has a predetermined maximum capacity of goods as well as a starting terminal in which duty of the vehicle starts. Moreover, a start time is assigned to each vehicle indicating the time that the vehicle leaves its starting terminal. The vehicle must leave its start terminal at the starting time, even if a possible waiting time at the first node visited occurs. The goal is to construct valid routes for each vehicle, such that time windows and capacity constraints are satisfied along each route, each pickup is served before the corresponding delivery, pickup and deliveries of each call are served on the same route and each vehicle only serves calls it is allowed to serve. The construction of the routes should be in such a way that they minimize the cost function. There is also a compatibility constraint between the vehicles and the calls. Thus, not all vehicles are able to handle all the calls. If we are not able to handle all calls by our fleet, we have to outsource them and pay the cost of not transporting them. For more details, readers are referred to Hemmati, Hvattum, Fagerholt, & Norstad (2014).

5. Experimental setup

In this section, we explain the baseline methods, process of hyperparameter selection, and dataset generation methods used for evaluation of the DRLH framework.

5.1. Experimental environment

The computational experiments in this paper were run on a desktop computer running a 64-bit Ubuntu 20.04 operating system with a AMD Ryzen 5 3600 processor and 32GB RAM.

5.2. Baseline models

Four baseline frameworks are chosen to compare with DRLH. Three of these methods use the same approach as DRLHin selecting a heuristic from the same set of heuristics at each iteration with the difference being in selection strategy. The last baseline uses a trained Deep RL agent to build a route by selecting a node at each step. The details of the baselines are presented in the following.

5.2.1. Adaptive large neighborhood search (ALNS)

As our approach is improving on the ALNS algorithm, this method is chosen as a baseline for performance comparison. This framework measures the performance of each heuristic using a scoring function for a certain number of iterations, referred to as a *segment*. At the end of each segment, the probability of choosing a heuristic during the next segment is updated using the aggregated scores of each heuristic in the previous segment. The extent to which the scores of the previous segment should contribute to updating the weights is controlled by the *reaction factor*.

There is a trade-off between speed and stability when choosing the values of the segment size and the reaction factor. Longer segments mean less frequent updates of the weights, but may increase the quality of the update. Similarly, a low reaction factor means that the weights can take longer to reach their desired values, but may also prevent sudden unfavorable changes to the weights due to the stochastic nature of the problem.

5.2.2. Uniform random selection (URS)

As a simpler approach to the selecting heuristics in each iteration, this method selects the heuristic randomly from H with equal probabilities.

5.2.3. Tuned random selection (TRS)

We introduce another baseline to our experiments which is refered to as TRS. For this method, we tuned the probabilities of selecting heuristics using the method of IRace (López-Ibáñez, Dubois-Lacoste, Pérez Cáceres, Birattari, & Stützle, 2016). The package "IRace" applies iterative F-Race to tune a set of parameters in an optimization algorithm (heuristic probabilities in our method) based on the performance on the training dataset.

5.2.4. Attention Module (AM) based Deep RL heuristic

We also consider the AM method of Kool et al. (2019) which achieved state-of-the-art results among the Deep RL based method for solving combinatorial optimization problems. This method uses the Deep RL agent combined with deep attention representation learning to build the solution at each step in a constructive manner using problem specific features from the environment. As a result, when applied on new problems, a new set of features as well as a problem specific representation learning scheme need to be defined. For example, the time window and vehicle incompatibility constraints were not mentioned in the original paper and for that reason we can not solve the difficult problem of PDPTW with this framework.

5.3. Hyperparameter selection

The hyperparameters for the Deep RL agent determine the speed and stability of the training process and also the final performance of the trained model. A small learning rate will cause training to take longer, but the smaller updates to the neural network also increase the chance of a better final performance once the model has been fully trained. Because the training process is done in advance of the testing stage, we opt for a slow and stable approach in order to train the best models possible. The hyperparameters of Deep RL agent for the experiments are listed in Table 4.

In order to decide on the hyperparameters for DRLH, some initial experiments were performed on the PDP problem (as the simple baseline problems compared to others) on a separate validation set to see which combinations performed best. The resulting set of hyperparameters have been applied for all experiments in this paper. Our motivation for doing so is that we wanted to test the generalizability of the framework in terms of the hyperparameters as well as the performance on different problems. By tuning the hyperparameters on a simpler problem and applying them to all other problems of all sizes and variations, we tried to avoid overtuning DRLH for every separate problem to keep the evaluation fair for the baseline methods and make sure that the advantage of our approach is in the decision making approach not the choice of hyperparameters for each problem. Moreover, this adds to the generalizability trait of the framework that does not require hyperparameter selection for each specific problem. Based on our experiments we found that these set of hyperparameters work very well across all the problem variations that we tested. It is likely that these hyperparameters can work for any underlying combinatorial optimization problem, as the hyperparameters for DRLH are related to the high-level problem of *heuristic selection*, which stays the same, regardless of what the underlying combinatorial optimization problem actually is. In the case of ALNS, we apply the same set of optimized hyperparameters that are suggested by Hemmati et al. (2014), which is optimized for solving the benchmark of PDPTW.

5.4. Dataset generation

For all the problem variations we generate a distinct training set consisting of 5000 instances, and a distinct testing set consisting of 100 instances. Additionally, for PDPTW we also utilize a known set of benchmark instances for testing (Hemmati et al., 2014).

5.4.1. CVRP

CVRP data instances are generated in accordance with the generation scheme of Nazari et al. (2018), Kool et al. (2019), but we also add two bigger problem variations. Instances of sizes $N = 20$, $N = 50$, $N = 100$, $N = 200$ and $N = 500$ are generated where N is the number of orders. For each instance the depot location and node locations are sampled uniformly at random from the unit square. Additionally, each order has a size associated with it defined as $\hat{\gamma} = \gamma_i / D_N$ where γ_i is sampled from the discrete set of $\{1, \dots, 9\}$, and the normalization factor D_N is set as $D_{20} = 30$, $D_{50} = 40$, $D_{100} = 50$, $D_{200} = 50$, $D_{500} = 50$, for instances with N orders, respectively.

5.4.2. PJSP

For the PJSP we generate instances of sizes $N = 20$, $N = 50$, $N = 100$, $N = 300$ and $N = 500$ where N is the number of jobs and using $M = \lfloor N/4 \rfloor$ machines. Job i's required processing steps PS_i are sampled from the discrete set of $\{100, 101, \dots, 1000\}$, and machine m's speed S_m, in processing steps per time unit, is sampled from $\mathcal{N}(\mu, \sigma^2)$ with $\mu = 10$, $\sigma = 30$, and the speed is rounded to the nearest integer and bounded to be at least 1. From there we get that the time required to process job i on machine m is calculated as $\lceil PS_i / S_m \rceil$.

5.4.3. PDP

For this problem, PDP data instances of sizes $N = 20$, $N = 50$, and $N = 100$ are generated where N is the number of nodes based on the generation scheme of Nazari et al. (2018), Kool et al. (2019). For each instance the depot location and node locations are sampled uniformly at random in unit square. Half of the nodes are pickup locations whereas the other half is the corresponding delivery locations. Additionally, each call has a size associated with it defined as $\hat{\gamma} = \gamma_i / D_N$ where γ_i is sampled from the discrete set of $\{1, \dots, 9\}$, and the normalization factor D_N is set as $D_{20} = 15$, $D_{50} = 20$, $D_{100} = 25$, for each problem with N number of nodes respectively.

5.4.4. PDPTW

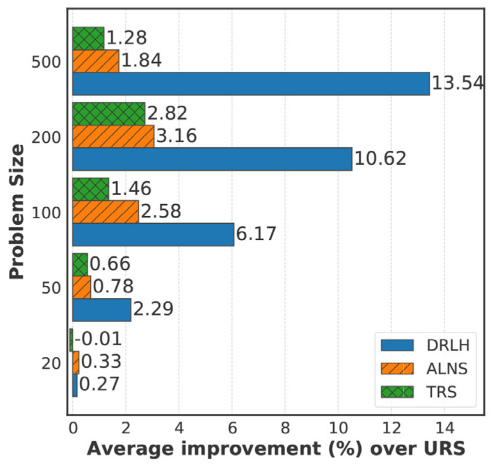
For the PDPTW we use instances with different combinations of number of calls and number of vehicles, see Table 5. For generating the training set and the 100 test instances, we use the provided instance generator of Hemmati et al. (2014). Additionally, we

Table 4
The hyperparameters used during training for the Deep RL agent of DRLH.

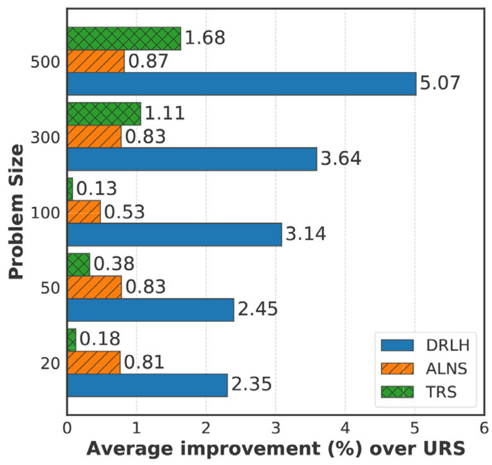
Hyperparameter	Value
Learning rate	1e−5
Batch size	64
First hidden layer size	256
Second hidden layer size	256
Discount factor	0.5

Table 5
Properties of different variations of the PDPTW instance types.

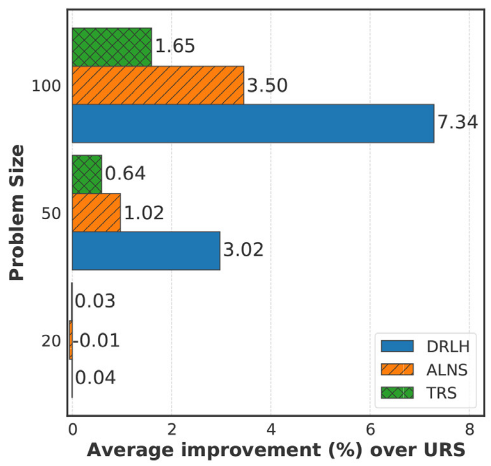
#Calls	#Vehicles	#Vehicle types
18	5	3
35	7	4
80	20	2
130	40	2
300	100	2



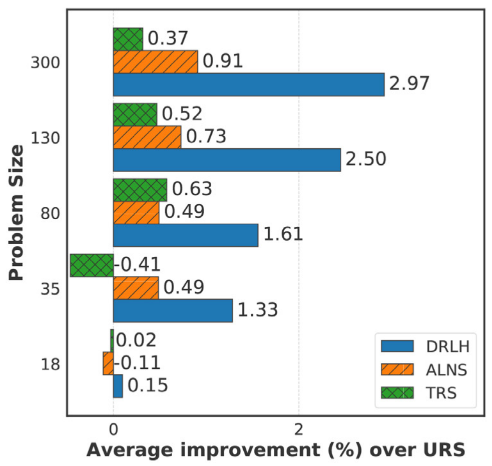
(a) CVRP results, 1k iterations



(b) PJSP results, 1k iterations



(c) PDP results, 1k iterations



(d) PDPTW results, 1k iterations

Fig. 1. Performance of DRLH on the generated test set.

use benchmark instances of Hemmati et al. (2014) for the remaining results. The benchmark test set consists of some instances of each variation, which are solved 10 times during testing in order to calculate the average best objective for each instance. Previous work by Homsi, Martinelli, Vidal, & Fagerholt (2020) have found the global optimal objectives for these instances, and we use these optimal values in order to calculate the *Min Gap (%)* and *Avg Gap (%)* to the optimal values for instances with 18, 35, 80 and 130 calls. Additionally, we also generate and test on a much larger instance size of 300 calls where we do not have the exact global optimal objectives, but instead use the best known values found by DRLH with 10,000 iterations to calculate the *Min Gap (%)* and *Avg Gap (%)*.

6. Results

In this section, we present the results of different experiments on the performance of DRLH. In the first experiment (Section 6.1), we set the number of iterations of the search to 1000 to compare the quality of the best found objective by each algorithm over a limited number of iterations for different problem sizes in the test set. In the next experiment (Section 6.2), we increase the number of iterations for all the methods and compare their performance when enough iterations are provided to fully explore the problem space. We also report the results on the benchmark of Hemmati et al. (2014) instances (Section 6.3). In order to demonstrate another advantage of using DRLH, we conduct an experiment with increased number of heuristics to illustrate the dependence of each framework on the performance of individual heuristics when the number of heuristics exceeds a certain number (Section 6.4). Additionally, we report the convergence speed and the training and inference time of each framework on instances of each problem (sections 6.5 and 6.6). Next, to gain insight into the reason behind the superiority of DRLH compared to the state of the art, we provide some figures and discuss the difference in strategy behind choosing a heuristic between DRLH and ALNS (Section 6.7). Finally, we compare the performance of DRLH, with a Deep RL heuristic approach (Section 6.8). Additional experiments and results regarding the reward function, convergence speed, and dependency of DRLH on the size of the problem can be found in Appendix.

6.1. Experiment on generated test set

For this experiment, each method was evaluated on a test set of 100 generated instances for each of the problems introduced in Section 4. Figure 1(a) shows the improvement in percentage that

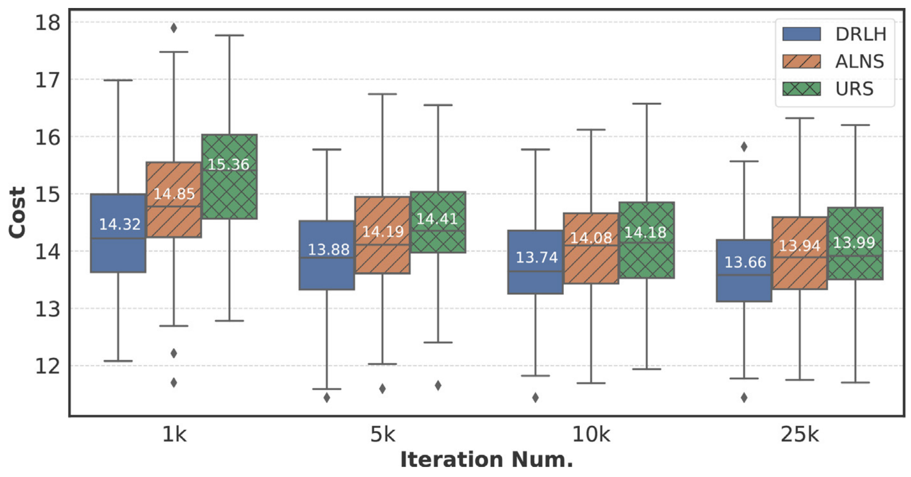


Fig. 2. Boxplot results for different iterations of PDP100.

Table 6
Average results for PDPTW instances with mixed call sizes after 1000 iterations.

		DRLH			ALNS			URS		
#C	#V	Min Gap (%)	Avg Gap (%)	Time (s)	Min Gap (%)	Avg Gap (%)	Time (s)	Min Gap (%)	Avg Gap (%)	Time (s)
18	5	**0.00**	**0.18**	32	0.00	0.46	25	0.00	0.40	12
35	7	2.67	**5.78**	9	3.45	7.08	36	**2.46**	6.40	27
80	20	**3.04**	**4.85**	37	3.64	6.51	98	4.62	7.23	100
130	40	**3.44**	**4.66**	100	4.00	6.24	186	4.85	6.71	176
300	100	**2.40**	**3.15**	637	3.10	5.04	599	5.29	6.51	398

using DRLH, ALNS, and TRS have over using URS on CVRP instances of different sizes. We see that DRLH is able to outperform all the baselines for all the instance sizes except for the smallest size. There is also a clear trend that shows how DRLH becomes increasingly better compared to other methods on larger instance sizes. Figure 1(b) shows a similar result for the PJSP problem. We see that DRLH is able to outperform the other methods for all of the instance sizes tested. Compared to the previous results, we see that the degree of improvement on larger instance sizes is less prominent for DRLH, but we also see that ALNS does not perform noticeably better on larger instance sizes at all. Because of that we still see a clear separation in performance between DRLH and ALNS on larger instance sizes that seem to grow with larger instance sizes. Finally, we observe a similar trend for PDP and PDPTW as for the other problems, which can be seen in Fig. 1(c) and (d), respectively. From this figure we see that DRLH outperforms ALNS and URS on all instance sizes tested and that performance difference tends to increase with larger instance sizes.

6.2. Experiment on increased number of iterations

Figure 2 shows that the number of iterations for improving the solution affects the minimum costs found for all the methods. We see that DRLH outperforms the baselines when tested for 1000, 5000, 10,000 and 25,000 iterations, and that the percentage difference between DRLH, ALNS and URS gets smaller as the number of iterations grows larger. Intuitively this makes sense as all three methods are getting closer to finding the optimal objectives for the test instances, and more iterations for improving the solution during the search makes the choices of which heuristics to select less sensitive compared to searching for a smaller number of iterations.

6.3. Experiment on the PDPTW benchmark dataset

In this section, we report results for PDPTW on the benchmark test set shown in Tables 6, 7 and 8 for 1000, 5000 and 10,000 iterations, respectively. We see from the tables that DRLH outperforms ALNS and URS on all of the tests on average, showing that it can find high quality solutions and has a robust average performance. Furthermore, we can see that the performance difference between DRLH and the baselines increases on bigger instances, meaning that DRLH scales favorably to the size of the problem, making it more viable for big industrial-sized problems compared to ALNS and URS.

We have also included the average time in seconds for optimizing the test instances. Note that the difference in time-usage is not directly dependent on the framework for selecting the heuristics (DRLH, ALNS, URS), but rather on the difference in time-usage of the heuristics themselves. This means that if all the heuristics used the same amount of time, then there would not be any time difference between the frameworks. However, because there is a relatively large variation in the time-usage between the different heuristics, we see a considerable variation between the frameworks as they all have different strategies for heuristic selection.

6.4. Experiment on the increased pool of heuristics

In addition to the set of heuristics mentioned in Section 3.1 we have also created an extended set of heuristics (see list in Appendix B). In total this extended set consists of 142 heuristics. Figure 3 shows the average gap of using the extended set compared to using the standard set for each of DRLH, ALNS and URS on PDPTW. The extended set obtains worse results on average compared to the standard set, but there is an interesting difference between the performance hit of DRLH, ALNS and URS when comparing the results of the extended set and the standard set. We see from Fig. 3 that DRLH is relatively unaffected by increasing the number of available heuristics (being only 0.02% worse on average), while ALNS and URS are performing much worse when using the extended set, and ALNS is hit especially hard. A likely reason for this is that there are too many heuristics to accurately explore

Table 7
Average results for PDPTW instances with mixed call sizes after 5000 iterations.

		DRLH			ALNS			URS		
#C	#V	Min Gap (%)	Avg Gap (%)	Time (s)	Min Gap (%)	Avg Gap (%)	Time (s)	Min Gap (%)	Avg Gap (%)	Time (s)
18	5	**0.00**	**0.00**	56	0.00	0.11	159	0.00	0.01	64
35	7	1.02	**2.95**	218	**0.78**	3.24	207	1.26	3.49	141
80	20	**1.76**	**3.25**	201	2.11	4.04	503	2.54	4.14	471
130	40	**2.10**	**3.14**	530	2.51	3.93	837	2.91	4.09	767
300	100	**0.48**	**1.15**	2580	1.01	2.35	2062	2.07	2.99	2352

Table 8
Average results for PDPTW instances with mixed call sizes after 10,000 iterations.

		DRLH			ALNS			URS		
#C	#V	Min Gap (%)	Avg Gap (%)	Time (s)	Min Gap (%)	Avg Gap (%)	Time (s)	Min Gap (%)	Avg Gap (%)	Time (s)
18	5	**0.00**	**0.00**	219	0.00	0.02	338	0.00	0.00	102
35	7	**0.67**	**2.02**	182	0.78	2.66	410	0.68	2.77	289
80	20	**1.80**	**2.95**	321	2.03	3.33	757	2.17	3.36	972
130	40	**1.93**	**2.84**	877	2.38	3.34	1307	2.56	3.37	1609
300	100	**0.00**	**0.64**	4630	0.55	1.89	4120	1.46	2.18	4203

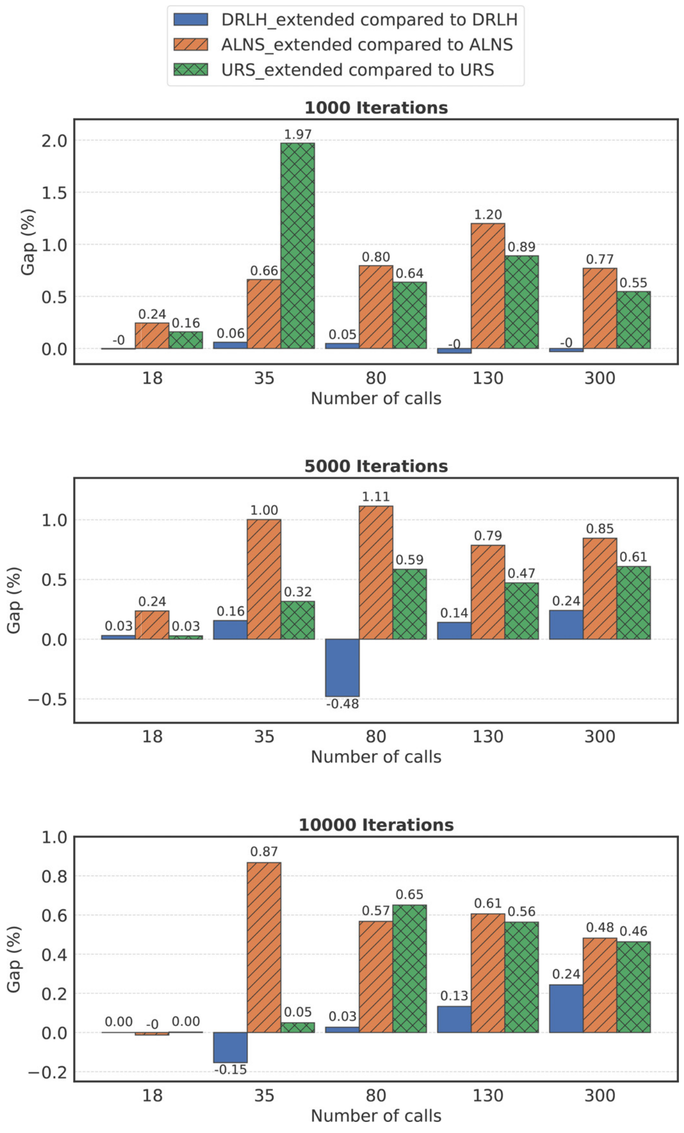


Fig. 3. Results of an Increased Pool of Heuristics.

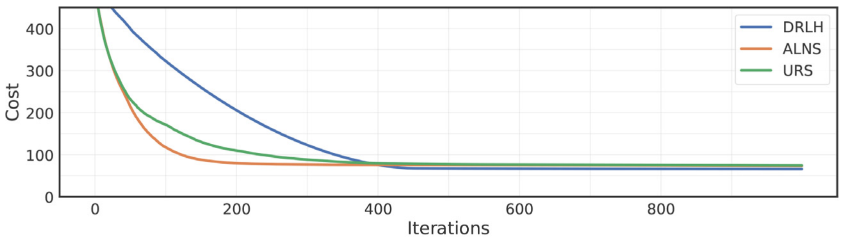
all of them during the search in order to identify the best heuristics and take advantage of them during the search.

An important conclusion from this result (albeit one that needs further empirical proof) is that when using DRLH, it is possible to supply it with a large number of heuristics and let DRLH identify the best ones to use. This is not possible for ALNS and consequently it is often necessary to spend time carrying out prior experiments with the aim of finding a small set of the best performing heuristics to include in the final ALNS model. This also resonates with the conclusion of Turkeš et al. (2021), who argue that the performance of ALNS benefits more from a careful a priori selection of heuristics, than from an elaborate adaptive layer. Considering that prior experiments can be quite time consuming, using DRLH can lead to a simpler development phase where heuristics can be added to DRLH without needing to establish their effectiveness beforehand, and not having to worry whether adding them will hurt the overall performance. Should a heuristic be unnecessary, then DRLH will learn to not use it during the training phase.

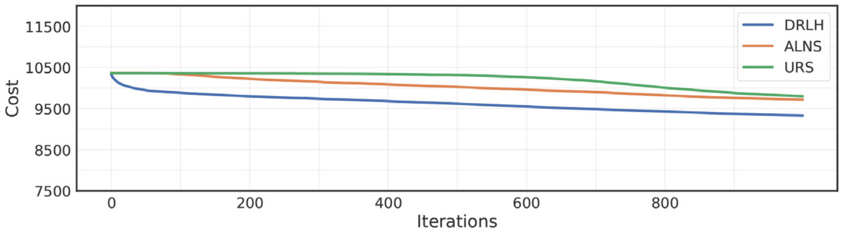
In addition to DRLH having a simpler development phase, an increased (or more nuanced) set of heuristics also has a larger potential to work well for a wide range of conditions, such as for different problems, instance sizes and specific situations encountered in the search. In other words, reducing the set of heuristics could negatively affect the performance of ALNS, but much less so for DRLH. Some heuristics work well only in specific situations, and so removing these “specialized” heuristics due to their poor average performance gives less potential for ALNS to be able to handle a diverse set of problem and instance variations compared to DRLH, which learns to take advantage of any heuristic that performs well in specific situations. Of course, these claims are based on a limited number of experiments and should be validated in a broad range of (future) experiments.

6.5. Average performance results

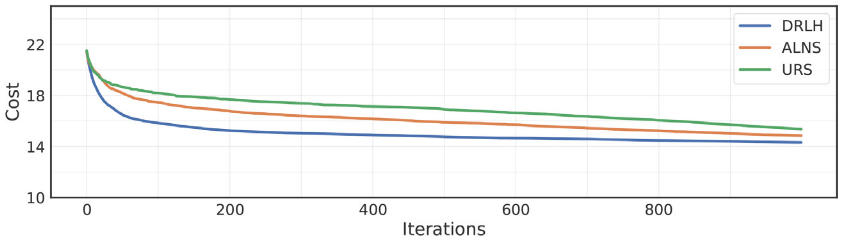
In this section, we explore the speed and characteristics of the performance of DRLH, ALNS and URS on the different problems. Fig. 4 shows that DRLH is able to quickly find better solutions compared to ALNS and URS for all the problems. Although for CVRP, DRLH takes a little bit longer initially, but ultimately reaches a much lower average minimum cost before the convergence of all three methods start to stagnate. For all the problems, DRLH is able



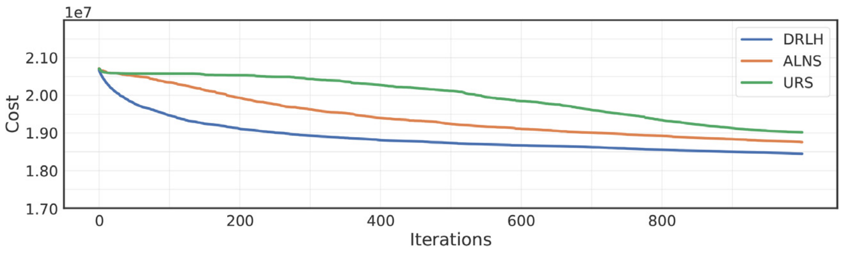
(a) CVRP, 500, 1k



(b) PJSP, 500, 1k



(c) PDP, 100, 1k



(d) PDPTW, 300, 1k

Fig. 4. Average performance of DRLH, ALNS and URS on each of the problems.

to reach a better cost after less than 500 iterations than what ALNS is able to reach after 1000 iterations. With the exception of the CVRP problem, DRLH is also extremely efficient in the beginning of the search, reaching costs in only 100 iterations that takes ALNS approximately 500 iterations to match. We refer to Appendix C for a complete collection of performance plots for all the problems that we have tested.

6.6. Training and inference time needed for each problem

Tables 9 and 10 report the time needed for training and solving for the instances of each problem, respectively. The main difference between DRLH and the baselines is the approach to decision making when it comes to choosing the next heuristic. This decision making process, on itself, does not add much overhead on the computational time of the methods. The main difference in the speed of these methods is the speed of the operators that they choose. This means that in some cases DRLH chooses operators that are faster or slower compared to baseline which results in lower or higher computational time. Therefore, when it comes to computational time, there is not much difference between these methods. This can also be shown in Table 10, in which in some cases DRLH is faster than the other two baselines and in some cases it is slower. It should be noted that the execution time of the operators can be improved if implemented carefully or using a faster programming language, e.g., C. However, the main focus of the paper is to improve the hyperheuristic approach of choosing the next heuristic at each step, not the execution time.

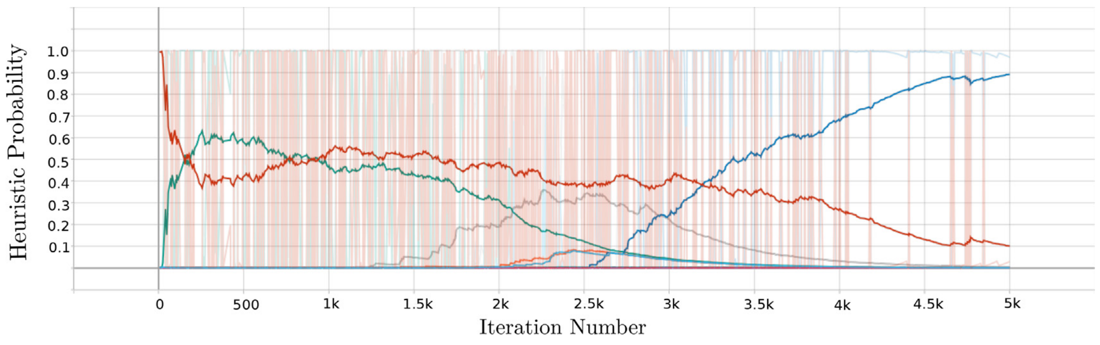
6.7. Comparison between heuristic selection strategies

Figure 5 demonstrate the probability of selecting heuristics at each step of the search for DRLH and ALNS in which each line corresponds to the probability of one heuristic at every step of the search. The “micro-level” heuristic usage of DRLH means that DRLH is able to drastically change the probabilities of selecting heuristics from one iteration to the next by taking advantage of the information provided by the search state, see Fig. 5(a) and (b). This is in contrast to the “macro-level” heuristic usage of ALNS where the probabilities of selecting operators only are updated at the beginning of each segment, meaning that the decision

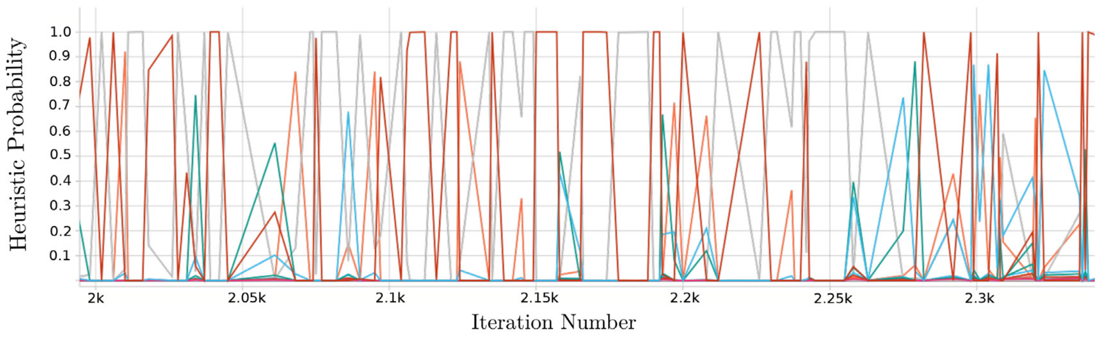
Table 9
Training time for DRLH on different problems.

Problem	Size	#Iterations	#Training Instances	Total training time (s)	Average time per instance (s)
CVRP	20	1k	1000	4586.85	4.59
CVRP	50	1k	1000	12394.3	12.39
CVRP	100	1k	1000	36330.0	36.33
CVRP	200	1k	100	8618.64	86.19
CVRP	500	1k	50	26483.2	529.66
PJSP	20	1k	1000	28233.7	28.23
PJSP	50	1k	1000	35552.1	35.55
PJSP	100	1k	500	16576.8	33.15
PJSP	300	1k	100	19758.1	197.58
PJSP	500	1k	100	79975.3	799.75
PDP	20	1k	500	1868.66	3.74
PDP	50	1k	100	2160.65	21.61
PDP	100	1k	100	12875.3	128.75
PDPTW	18	1k	600	25340.2	42.23
PDPTW	35	1k	600	12154.9	20.26
PDPTW	80	1k	500	20704.4	41.41
PDPTW	130	1k	100	8595.9	85.96
PDPTW	300	1k	90	53657.5	596.19

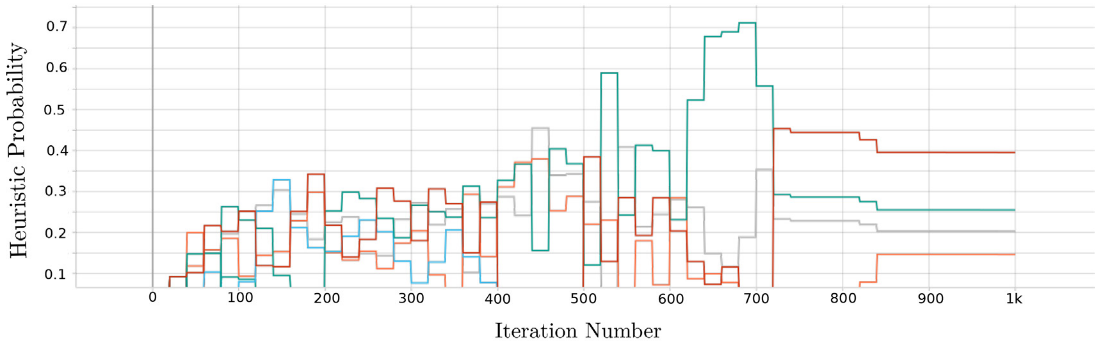
Deep Reinforcement Learning Hyperheuristic



(a) Smoothed probabilities of selecting heuristics for DRLH.



(b) Actual probabilities of selecting heuristics for DRLH, zoomed in between iteration 2000-2300.



(c) Actual probabilities of selecting heuristics for ALNS.

Fig. 5. Example of the probability of selecting heuristics for DRLH and ALNS.

Table 10
Average Time (seconds) required for solving the test instances for each method DRLH, ALNS and URS.

Problem	Size	#Iterations	DRLH	ALNS	URS
CVRP	20	1k	4.08	11.58	7.75
CVRP	50	1k	11.58	35.17	23.52
CVRP	100	1k	34.28	99.58	50.65
CVRP	200	1k	102.76	221.22	94.07
CVRP	500	1k	621.74	664.54	238.86
PJSP	20	1k	20.37	18.06	5.65
PJSP	50	1k	30.69	41.84	15.9
PJSP	100	1k	57.05	76.15	34.0
PJSP	300	1k	199.37	237.58	110.81
PJSP	500	1k	453.92	462.34	195.67
PDP	20	1k	3.89	4.17	1.85
PDP	50	1k	31.4	20.15	9.93
PDP	100	1k	159.86	79.58	45.86
PDPTW	18	1k	32.61	23.85	9.18
PDPTW	35	1k	10.67	29.75	21.18
PDPTW	80	1k	34.82	71.27	68.33
PDPTW	130	1k	110.67	139.45	132.32
PDPTW	300	1k	500.9	438.65	361.39

making of ALNS within a single segment is random according to the locked probabilities for that segment, see Fig. 5(c). Depending on the problem and available heuristics to select, there might exist exploitable strategies and patterns for heuristic selection, such as heuristic(s) that: work well when used together, work well for escaping local minima, work well on solutions not previously encountered during the search. Using DRLH, these types of exploitable strategies can be automatically discovered without the need for specially tailored algorithms designed by human experts. We refer to one such exploitable strategy found by DRLH on our problems with our provided set of heuristics as minimizing “wasted actions”. We define a wasted action as the selection of a deterministic heuristic (in our case *Find_single_best*) for two consecutive unsuccessful iterations. The reason that this action is “wasted” is because of the deterministic nature of the heuristic, which makes it so that if the solution did not change in the previous iteration, then it is guaranteed not to change in the following iteration as well. Even though we have not specifically programmed DRLH to utilize this strategy, it becomes clear by examining Table 11 that the DRLH has picked up on this strategy when learning to optimize micro-level heuristic selection. Table 11 shows that the number of wasted actions for DRLH is almost non-existent for most problem variations. ALNS on the other hand ends up with far more wasted actions than DRLH, even though ALNS also uses *Find_single_best* much more seldom on average. Figure 5(c) shows how the heuristic probabilities for ALNS remain locked within the segments, making it impossible for ALNS to exploit strategies such as minimizing wasted actions which relies on excellent micro-level heuristic selection such as what DRLH demonstrates.

6.8. Performance comparison with AM deep RL heuristic

For this experiment, we ran the AM method of Kool et al. (2019) on our test datasets for the CVRP problem. The trained models and the implementation of the models needed to solve the problem have been provided publicly by the authors of this paper. The dataset generation procedure for both our work and the AM paper follow the work of Nazari et al. (2018). As a result, the models are well fit to be evaluated on our test set. For their method we considered three different approaches : *Greedy, Sample_128* and *Sample_1280*. In the greedy approach, at each step the node with the most probability is chosen. In the sampling approach, 128 and 1280 different solutions are sampled based on the probability of each node at each step. We test these methods for sizes $n = 20, 50, 100$ of the CVRP problem. The time and resources (Graphical Processing Units) needed to train the AM method for sizes larger than 100 scales exponentially due to heavy calculations needed for their representation learning method. Therefore, we only solve this problem for the mentioned instance sizes.

Figure 6 illustrates the comparison of performance of our method with the AM method of Kool et al. (2019). As shown in

Table 11
The percentage of wasted actions of the total number of deterministic heuristics selected, averaged over the test set for each problem.

(a) CVRP

		Wasted Actions (%)	
#Orders	#Iterations	DRLH	ALNS
20	1k	3.37	26.55
50	1k	0.00	23.98
100	1k	1.22	19.48
200	1k	0.00	23.43
500	1k	0.01	25.15

(b) PJSP

		Wasted Actions (%)	
#Jobs	#Iterations	DRLH	ALNS
20	1k	0.00	20.82
50	1k	0.86	24.57
100	1k	0.00	24.80
300	1k	0.00	24.85
500	1k	0.00	24.50

(c) PDP

		Wasted Actions (%)	
#Calls	#Iterations	DRLH	ALNS
20	1k	6.82	31.53
50	1k	0.00	29.00
100	1k	0.00	28.01
100	5k	0.02	30.62
100	10k	0.00	33.86
100	25k	0.00	32.69

(d) PDPTW

		Wasted Actions (%)	
#Calls	#Iterations	DRLH	ALNS
18	1k	0.00	21.68
35	1k	0.00	28.65
80	1k	0.00	24.50
130	1k	0.00	19.60
300	1k	0.00	17.90
18	5k	0.00	30.88
35	5k	0.00	36.26
80	5k	0.00	27.49
130	5k	0.00	26.98
300	5k	0.00	26.10
18	10k	0.25	37.82
35	10k	0.00	36.60
80	10k	0.00	32.41
130	10k	0.08	29.67
300	10k	0.00	26.10

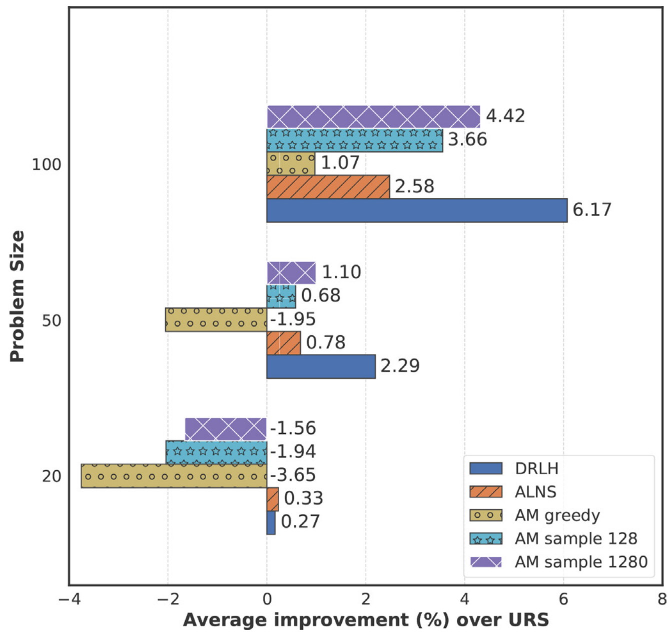


Fig. 6. Comparison of DRLH with the Deep RL method of Kool et al. (2019) (AM) on test instances of CVRP.

the figure, AM is not able to outperform the baseline of URS in the size 20 with any of the sampling methods. Regarding size 50 of the problem, in the greedy approach it still falls behind URS. However, given enough samples, AM manages to perform better than ALNS in some instances of the CVRP problem. On the other hand, our method of DRLH outperforms this approach in every single instance size as well as being able to handle different problems without any significant change in the code which is not the case for the method of Kool et al. (2019).

7. Concluding remarks

For quite some time now, it has increasingly become evident that the fields of machine learning and (heuristic) optimization can mutually benefit from an integration. On the one hand, recent advances in optimization can support the development of advanced machine learning methods, since these methods generally solve an optimization problem (e.g., what is the optimal subset of features from a data set that predict a certain outcome). This paper addressed the mirror issue: how can optimization approaches benefit from an integration of machine learning methods. We demonstrated that applying a well-known machine learning approach to the selection of low-level operators in a metaheuristic framework results in a robust mechanism that can be used to improve the performance of a heuristic on a broad range of optimization problems. We believe that approaches like the one presented in this paper have the potential to make the development of a powerful heuristic less dependent on the knowledge of an experienced developer with a deep insight into the structure of the specific problem being solved, and may therefore be instrumental in the integration of metaheuristics ideas into general purpose software packages. Our proposed DRLH, a general framework for solving combinatorial optimization problems, utilizes a trained Deep RL agent to select low-level heuristics to be applied on the solution in each iteration of the search based on a search state consisting of features from the search process. In our experiments, we solved four combinatorial optimization problems (CVRP, PJSP, PDP, and PDPTW) using our proposed approach and compared its performance with the baselines of ALNS and URS. Our results show that DRLH is able to select heuristics in a way that achieves better results in less number of iterations for almost all of the problem variations compared to ALNS and URS. Furthermore, the performance gap between DRLH and the baselines is shown to increase for larger problem sizes, making DRLH a suitable option for large real-world problem instances. Additional experiments on an extended set of heuristics show that DRLH is not negatively affected when selecting from a large set of available heuristics, while the performance of ALNS is much worse in this situation. Enriching or refining the state representation with additional information is possible with very little effort. We have experimented with adding problem-dependent information into the state representation and seen that this gives even better results than sticking with the simple chosen state representation. Yet once we start to introduce problem-dependent structure and constraint information into the state representation we lose some of the generality that we strive for with DRLH as we would have to separately engineer a different state representation for each new problem. For this reason we deem this outside of the scope of this paper and leave this area open for future work.

Future research should provide more empirical evidence for the superiority of DRLH over ALNS by applying this novel hyperheuristic to different problems. A potential direction for improving the model in the future is designing a reward function that is both stable and takes into account the difference of objective value at each iteration of the search. Initial experiments on alternative reward functions have shown promising results (see Appendix A), but are time-consuming to train and not very stable compared to the R^{5310} reward function that we have used in this paper.

Appendix A. Experiments on different reward functions

A1. R_t^{PM}

$$R_t^{PM} = \begin{cases} 1, & \text{if } f(x') < f(x) \\ -1, & \text{otherwise} \end{cases} \tag{A.1}$$

The R_t^{PM} reward function focuses more heavily on intensification by punishing any action choice that does not directly improve upon the current solution. This causes the agent to favor intensifying heuristics more strongly than R_t^{5310}. However, because the PPO framework leverages the *discounted future rewards* as opposed to only the immediate reward for training the agent, even the R_t^{PM} can cause the agent to select heuristics with a high likelihood of immediate negative reward if it sets it up for more positive rewards in future iterations.

Figure A.1 illustrates the distribution of minimum costs found on the PDP of size 100 test set after 1000 and 10,000 iterations for two different versions of DRLH, trained with reward functions R_t^{5310} and R_t^{PM} respectively. The model trained with R_t^{5310} achieves a lower median and quantile values for both iteration variations, compared to the model trained with R_t^{PM}. This makes the R_t^{5310} reward function more reliable to perform relatively better, and we therefore decided to use the R_t^{5310} reward function in this paper.

Table A.1
Average results for PDPTW instances with mixed call sizes after 1000 iterations.

		DRLH with R_t^{5310}		DRLH with R_t^{MC}	
#C	#V	Min Gap (%)	Avg Gap (%)	Min Gap (%)	Avg Gap (%)
18	5	0.00	0.18	**0.00**	**0.11**
35	7	2.67	5.78	**1.48**	**3.65**
80	20	**3.04**	4.85	3.15	**4.39**
130	40	3.44	4.66	**2.99**	**4.33**
300	100	2.40	3.15	**2.28**	**3.00**

Table A.2
Average results for PDPTW instances with mixed call sizes after 10,000 iterations.

		DRLH with R_t^{5310}		DRLH with R_t^{MC}	
#C	#V	Min Gap (%)	Avg Gap (%)	Min Gap (%)	Avg Gap (%)
18	5	**0.00**	**0.00**	0.00	0.13
35	7	0.67	**2.02**	**0.42**	2.32
80	20	**1.80**	**2.95**	2.55	3.87
130	40	**1.93**	**2.84**	2.20	3.04
300	100	**0.00**	**0.64**	1.12	1.88

A2. R_t^{MC}

$$R_t^{MC} = \left\{ \frac{f(x_{best}) - f(x')}{f(x_{best})} \right. \tag{A.2}$$

The R_t^{MC} is a reward function that more directly correlates with the intended objective of minimizing the cost of the best found solution, and to achieve this as quickly as possible. Instead of focusing on rewarding actions that directly improve the solution, this reward function is subject to the performance of the entire search process up to the current step, putting a greater emphasis on acting quickly and selecting heuristics that have a greater impact on the solution. The challenge with using this reward function compared to reward functions such as R_t^{5310} and R_t^{PM} is that there is an inherent delay between when a good heuristic is selected and when the reward function gives a good reward. This makes it more difficult to train an agent using this reward function, making training times much longer and less stable than with the R_t^{5310} reward function.

Having said that, the potential upside of using this reward function is very promising, and results in Table A.1 show that R_t^{MC} is able to outperform the R_t^{5310} reward function on 1k iteration searches. However, the agents were unable to learn effectively for larger number of iterations such as 10k (Table A.2), and so results for this shows that R_t^{MC} performs worse than R_t^{5310} on 10k iteration searches. A potential reason for why the R_t^{MC} agents were unable to learn well on 10k iteration searches is that the amount of improving iterations are much less frequent, making the feedback signal from the R_t^{MC} reward function even more delayed and high variance. Another potential reason is that the training required in order to solve 10k iteration searches likely needed more training than what was possible to carry out for our experiments due to time constraints with the experiments. We encourage future work on improving the integration of the R_t^{MC} reward function into the framework of DRLH as it likely has a lot of potential.

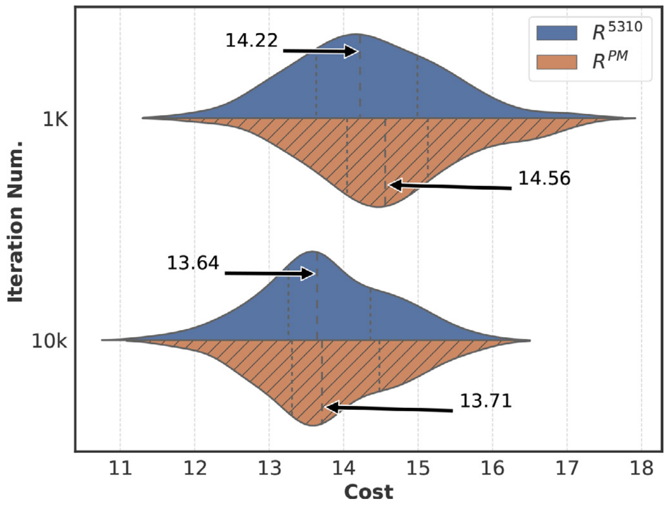


Fig. 7. Comparison of the two reward functions.

Appendix B. Extended set of heuristics

Tables A.3, A.4 and A.5 list the extended set of heuristics built up from 14 removal operators, 10 insertion operators and 2 additional heuristics, for a total of 14 × 10 + 2 = 142 total heuristics, using the generation scheme of Algorithm 2. Most of these heuristics only use problem-independent information, but some of them rely on problem-dependent information specific to the PDPTW problem.

Table A.3
List of extended removal operators.

Name	Description
Random_remove_XS	Removes between 2–5 elements chosen randomly
Random_remove_S	Removes between 5–10 elements chosen randomly
Random_remove_M	Removes between 10–20 elements chosen randomly
Random_remove_L	Removes between 20–30 elements chosen randomly
Random_remove_XL	Removes between 30–40 elements chosen randomly
Random_remove_XXL	Removes between 80–100 elements chosen randomly
Remove_largest_$\mathcal{D}$_S	Removes 5–10 elements with the largest $\mathcal{D}_i$
Remove_largest_$\mathcal{D}$_L	Removes 20–30 elements with the largest $\mathcal{D}_i$
Remove_τ	Removes a random segment of 2–5 consecutive elements in the solution
Remove_least_frequent_S	Removes between 5–10 elements that has been removed the least
Remove_least_frequent_M	Removes between 10–20 elements that has been removed the least
Remove_least_frequent_XL	
Remove_one_vehicle	Removes all the elements in one vehicle
Remove_two_vehicles	Removes all the elements in two vehicle

Table A.4
List of extended insertion operators.

Name	Description
Insert_greedy	Inserts each element in the best possible position
Insert_beam_search	Inserts each element in the best position using beam search
Insert_by_variance	Sorts the insertion order based on variance and inserts each element in the best possible position
Insert_first	Inserts each element randomly in the first feasible position
Insert_least_loaded_vehicle	Inserts each element into the least loaded available vehicle
Insert_least_active_vehicle	Inserts each element into the least active available vehicle
Insert_close_vehicle	Inserts each element into the closest available vehicle
Insert_group	Identifies the vehicles that can fit the most of the removed elements and inserts each elements into these
Insert_by_difficulty	Inserts each element using *Insert_greedy* ordered by their difficulty, which is a function of their compatibility with vehicles, strictness of time windows,size and more.
Insert_best_fit	Inserts each element into the vehicle that is the most compatible with the call.

Table A.5
List of extended additional heuristics.

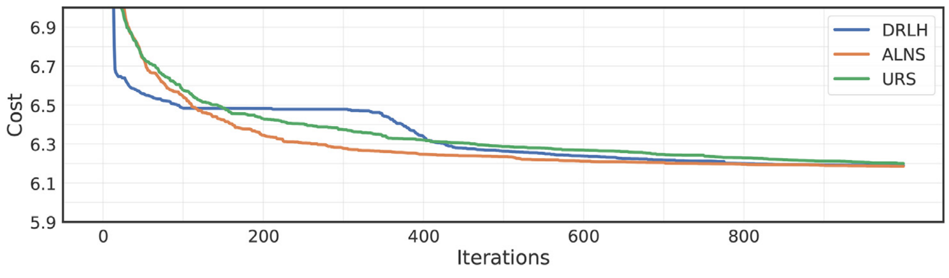
Name	Description
Find_single_best	Calculates the cost of removing each element and re-inserting it with *Insert_greedy*, and applies this procedure on the solution x for the element that achieves the minimum cost $f(x')$.
Rearrange_vehicles	Removes all of the elements from each vehicle and inserts them back into the same vehicles using *Insert_beam_search*

Appendix C. Additional performance plots

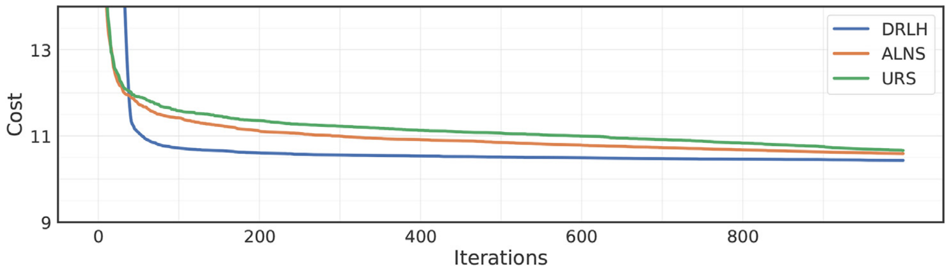
Figures 8, 9, 10 and 11 show the performance of DRLH, ALNS and URS averaged over the test set for all the problems that we have tested. These show that DRLH usually reaches better solutions more quickly than ALNS and URS, as well as ending up with better solutions overall.

Appendix D. Experiment on the cross size training scheme

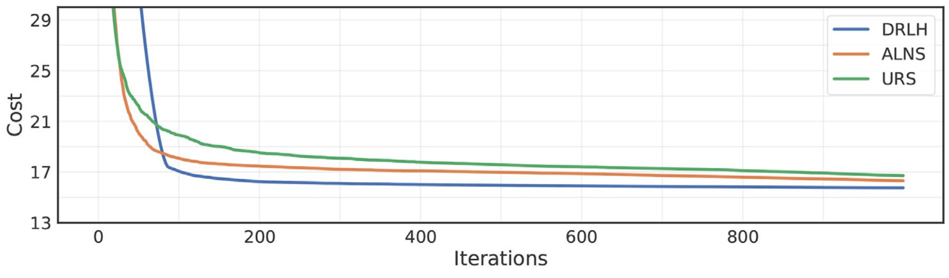
In this experiment, in the training phase, an instance of a specific problem with different size is solved by DRLH in each episode. This training scheme is referred to as Cross Size (CS) training. During test time, the trained model solved the test instances that were used in Section 6.1 as well as test instances of slightly different sizes that were seen during training. As seen in Fig. 12, it is possible to train one model that can handle many different variations of instance sizes quite well. Moreover, as shown in Fig. 12(e)–(h), the model does not specifically overfit on the specific instance sizes included in the training when evaluated on slightly different test data. This means that the DRLH_CS generalizes very well, even to sizes higher than any of the ones included in the training.



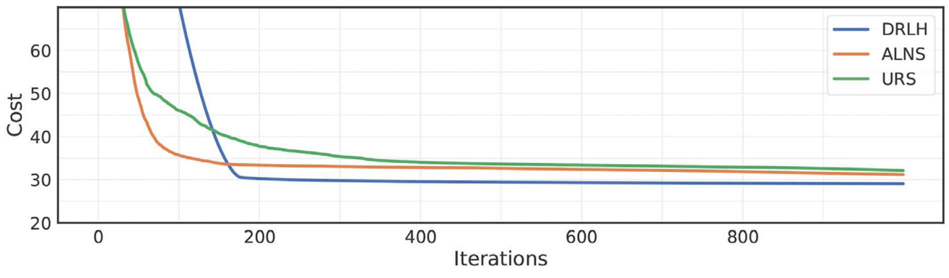
(a) CVRP, 20, 1k



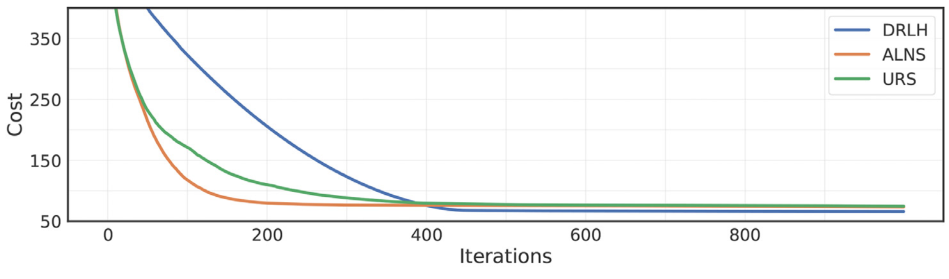
(b) CVRP, 50, 1k



(c) CVRP, 100, 1k

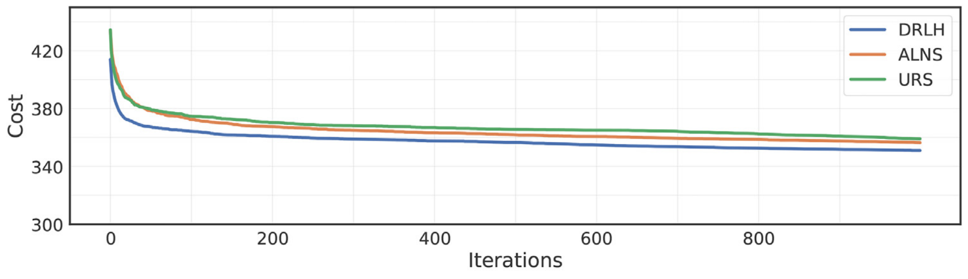


(d) CVRP, 200, 1k

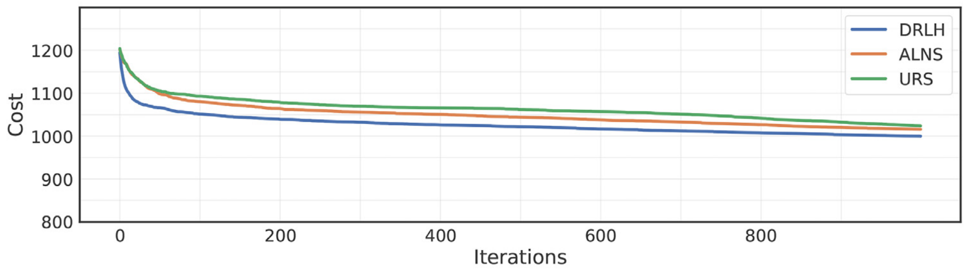


(e) CVRP, 500, 1k

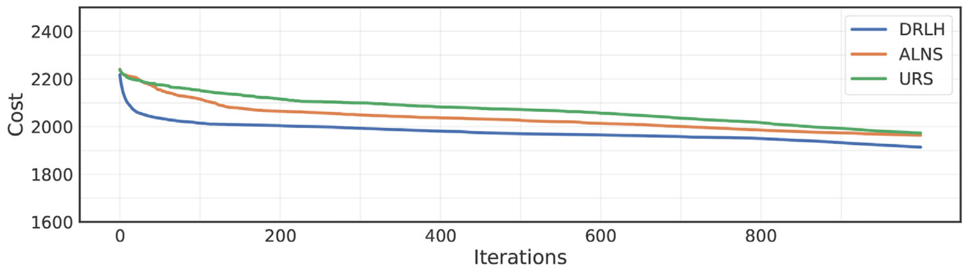
Fig. 8. Average performance of DRLH, ALNS and URS on CVRP.



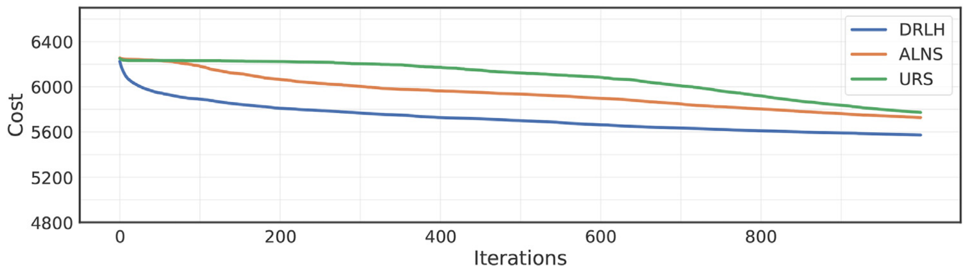
(a) PJSP, 20, 1k



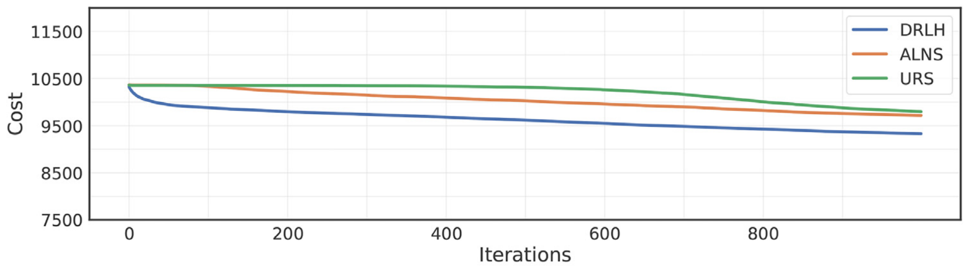
(b) PJSP, 50, 1k



(c) PJSP, 100, 1k

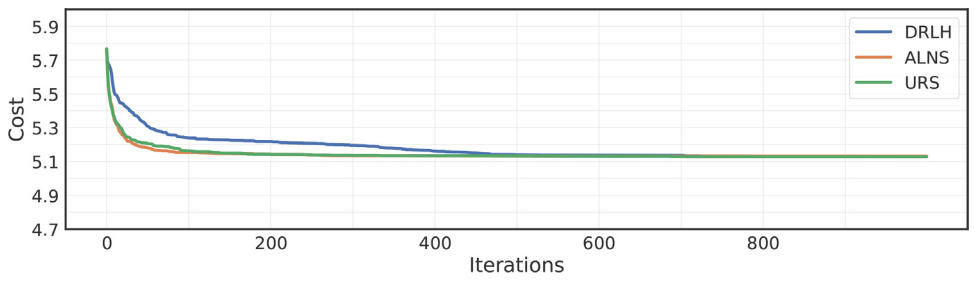


(d) PJSP, 300, 1k

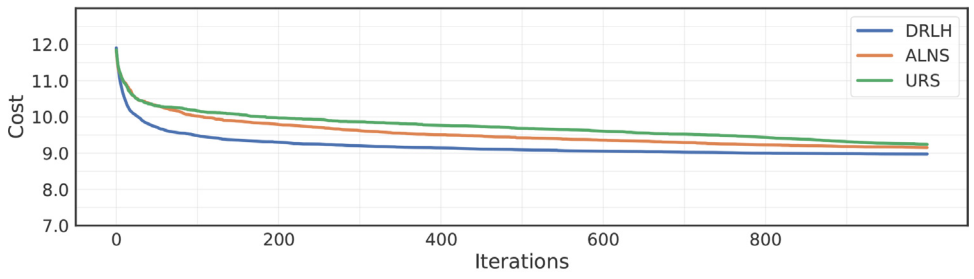


(e) PJSP, 500, 1k

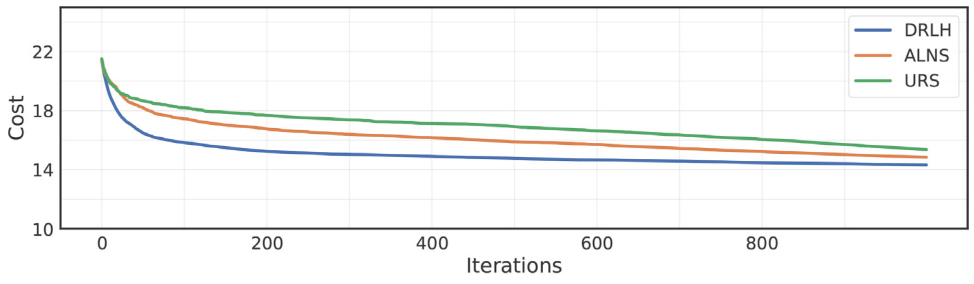
Fig. 9. Average performance of DRLH, ALNS and URS on PJSP.



(a) PDP, 20, 1k

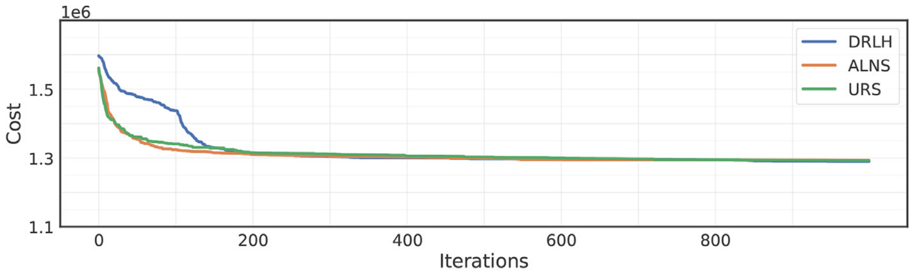


(b) PDP, 50, 1k

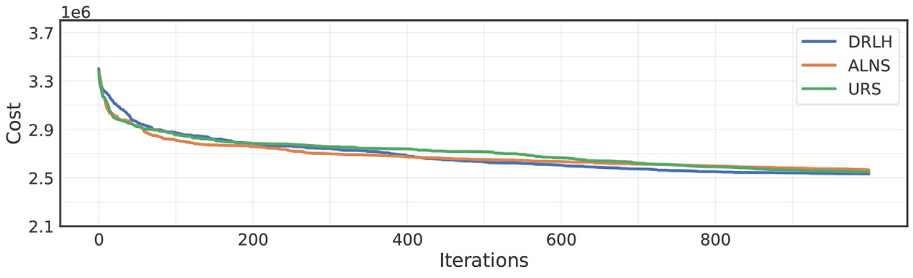


(c) PDP, 100, 1k

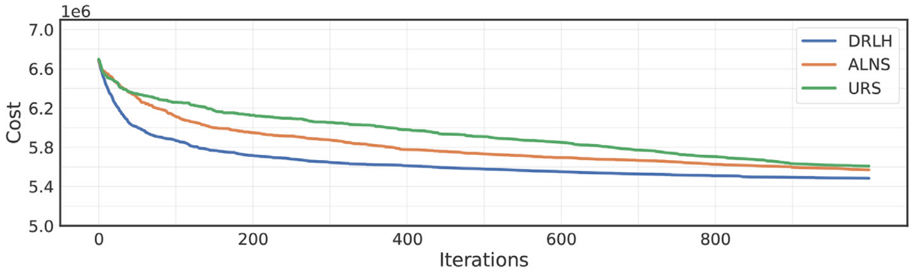
Fig. 10. Average performance of DRLH, ALNS and URS on PDP.



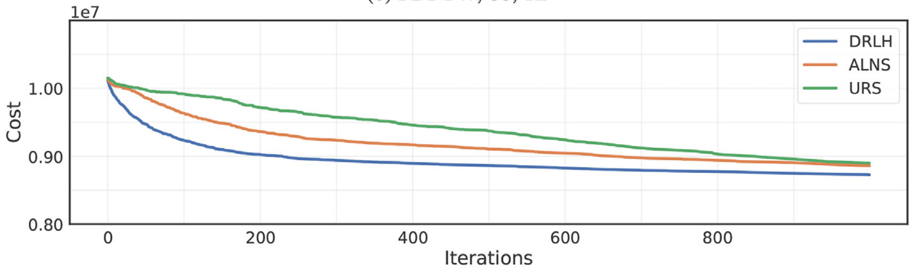
(a) PDPTW, 18, 1k



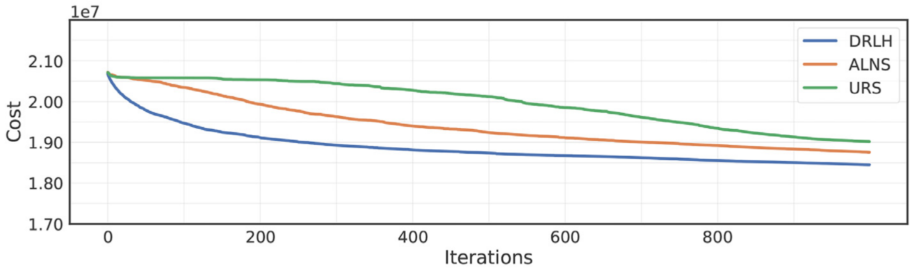
(b) PDPTW, 35, 1k



(c) PDPTW, 80, 1k

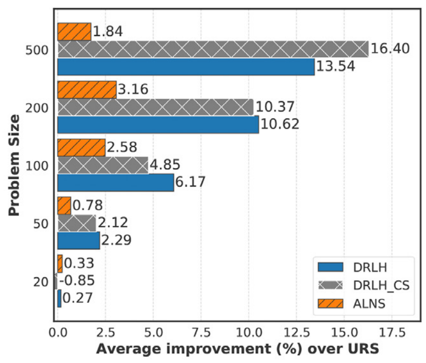


(d) PDPTW, 130, 1k

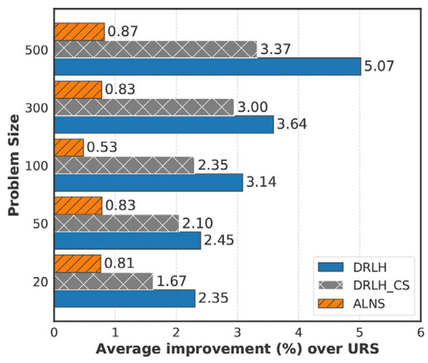


(e) PDPTW, 300, 1k

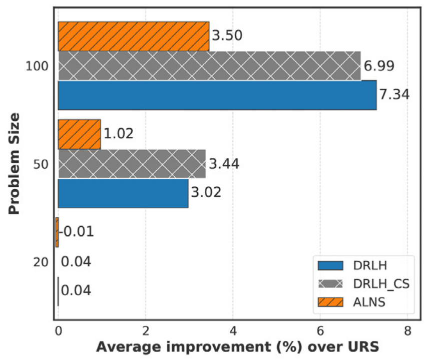
Fig. 11. Average performance of DRLH, ALNS and URS on PDPTW.



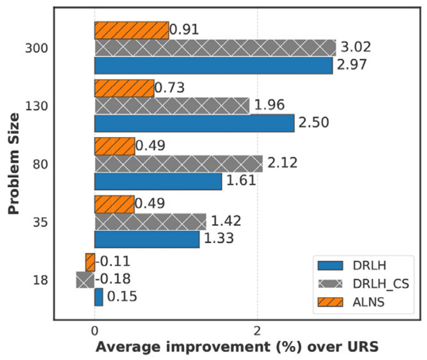
(a) CVRP results, same sizes as training



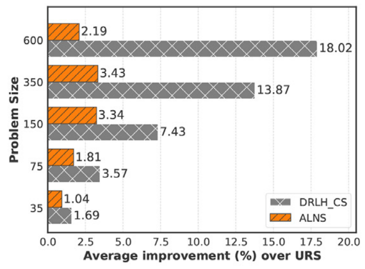
(b) PJSP results, same sizes as training



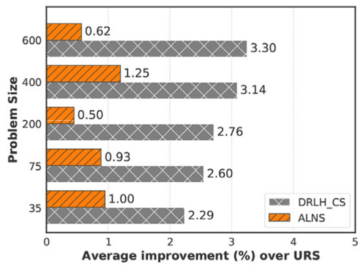
(c) PDP results, same sizes as training



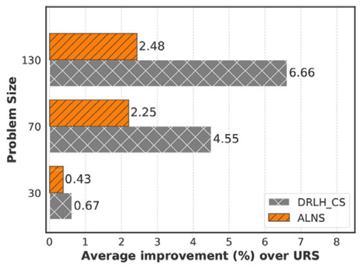
(d) PDPTW results, same sizes as training



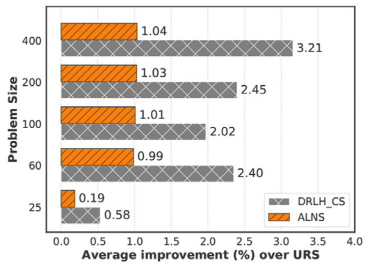
(e) CVRP results, unseen sizes in training



(f) PJSP results, unseen sizes in training



(g) PDP results, unseen sizes in training



(h) PDPTW results, unseen sizes in training

Fig. 12. Performance of DRLH with Cross Size (CS) training scheme on different problem sizes with 1k iterations.

References

Aksen, D., Kaya, O., Sibel Salman, F., & Özge Tüncel (2014). An adaptive large neighborhood search algorithm for a selective and periodic inventory routing problem. *European Journal of Operational Research, 239*(2), 413–426. https://doi.org/10.1016/j.ejor.2014.05.043.

Amodei, D., Olah, C., Steinhardt, J., Christiano, P. F., Schulman, J., & Mané, D. (2016). Concrete problems in AI safety. *CoRR*. http://arxiv.org/abs/1606.06565.

Asta, S., & Özcan, E. (2014). An apprenticeship learning hyper-heuristic for vehicle routing in hyflex. Orlando, Florida. https://doi.org/10.1109/EALS.2014.7009505.

Burke, E. K., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., & Woodward, J. R. (2010). *A classification of hyper-heuristic approaches*. In M. Gendreau, & J.-Y. Potvin (Eds.) (pp. 449–468). Boston, MA: Springer US.

Chen, C., Demir, E., & Huang, Y. (2021). An adaptive large neighborhood search heuristic for the vehicle routing problem with time windows and delivery robots. *European Journal of Operational Research, 294*(3), 1164–1180. https://doi.org/10.1016/j.ejor.2021.02.027.

Chen, X., & Tian, Y. (2019). Learning to perform local rewriting for combinatorial optimization. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems: vol. 32*. Curran Associates, Inc.. https://proceedings.neurips.cc/paper/2019/file/131f383b434fdf48079bff1e44e2d9a5-Paper.pdf

Cowling, P., Kendall, G., & Soubeiga, E. (2001). A hyperheuristic approach to scheduling a sales summit. In E. Burke, & W. Erben (Eds.), *Practice and theory of automated timetabling iii* (pp. 176–190). Berlin, Heidelberg: Springer Berlin Heidelberg.

Crama, Y., & Schyns, M. (2003). Simulated annealing for complex portfolio selection problems. *European Journal of Operational Research, 150*(3), 546–571. https://doi.org/10.1016/S0377-2217(02)00784-1. Financial Modelling

Demir, E., Bektaş, T., & Laporte, G. (2012). An adaptive large neighborhood search heuristic for the pollution-routing problem. *European Journal of Operational Research, 223*(2), 346–359. https://doi.org/10.1016/j.ejor.2012.06.044.

Dokeroglu, T., Sevinc, E., Kucukyilmaz, T., & Cosar, A. (2019). A survey on new generation metaheuristic algorithms. *Computers & Industrial Engineering, 137*, 106040. https://doi.org/10.1016/j.cie.2019.106040.

Friedrich, C., & Elbert, R. (2022). Adaptive large neighborhood search for vehicle routing problems with transshipment facilities arising in city logistics. *Computers & Operations Research, 137*, 105491. https://doi.org/10.1016/j.cor.2021.105491.

Goodfellow, I. J., Bengio, Y., & Courville, A. (2016). *Deep learning*. Cambridge, MA, USA: MIT Press. http://www.deeplearningbook.org

Grangier, P., Gendreau, M., Lehuédé, F., & Rousseau, L.-M. (2016). An adaptive large neighborhood search for the two-echelon multiple-trip vehicle routing problem with satellite synchronization. *European Journal of Operational Research, 254*(1), 80–91. https://doi.org/10.1016/j.ejor.2016.03.040.

Gullhav, A. N., Cordeau, J.-F., Hvattum, L. M., & Nygreen, B. (2017). Adaptive large neighborhood search heuristics for multi-tier service deployment problems in clouds. *European Journal of Operational Research, 259*(3), 829–846. https://doi.org/10.1016/j.ejor.2016.11.003.

Hemmati, A., & Hvattum, L. M. (2017). Evaluating the importance of randomization in adaptive large neighborhood search. *International Transactions in Operational Research, 24*(5), 929–942. https://doi.org/10.1111/itor.12273.

Hemmati, A., Hvattum, L. M., Fagerholt, K., & Norstad, I. (2014). Benchmark suite for industrial and tramp ship routing and scheduling problems. *INFOR: Information Systems and Operational Research, 52*(1), 28–38. https://doi.org/10.3138/infor.52.1.28.

Homsi, G., Martinelli, R., Vidal, T., & Fagerholt, K. (2020). Industrial and tramp ship routing problems: Closing the gap for real-scale instances. *European Journal of Operational Research, 283*(3), 972–990. https://doi.org/10.1016/j.ejor.2019.11.068.

Hottung, A., & Tierney, K. (2019). Neural large neighborhood search for the capacitated vehicle routing problem. *CoRR*. http://arxiv.org/abs/1911.09539.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science, 220*(4598), 671–680. https://doi.org/10.1126/science.220.4598.671.

Kool, W., van Hoof, H., & Welling, M. (2019). Attention, learn to solve routing problems!

Laborie, P., & Godard, D. (2007). Self-adapting large neighborhood search: Application to single-mode scheduling problems. In *Proceedings MISTA-07: Vol. 8*. Paris

Li, Y., Chen, H., & Prins, C. (2016). Adaptive large neighborhood search for the pickup and delivery problem with time windows, profits, and reserved requests. *European Journal of Operational Research, 252*(1), 27–38. https://doi.org/10.1016/j.ejor.2015.12.032.

Lu, H., Zhang, X., & Yang, S. (2020). A learning-based iterative method for solving vehicle routing problems. In *International conference on learning representations*. https://openreview.net/forum?id=BJe1334YDH

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., & Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives, 3*, 43–58. https://doi.org/10.1016/j.orp.2016.09.002.

Nazari, M., Oroojlooy, A., Snyder, L., & Takac, M. (2018). Reinforcement learning for solving the vehicle routing problem. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in neural information processing systems: vol. 31*. Curran Associates, Inc.. https://proceedings.neurips.cc/paper/2018/file/9fb4651c05b2ed70fba5afe0b039a550-Paper.pdf

Özcan, E., Misir, M., Ochoa, G., & Burke, E. (2010). A reinforcement learning - great-deluge hyper-heuristic for examination timetabling. *International Journal of Applied Metaheuristic Computing, 1*, 39–59.

Pisinger, D., & Ropke, S. (2019). Large neighborhood search. In *Handbook of metaheuristics* (pp. 99–127). Springer.

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

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*. abs/1707.06347. http://dblp.uni-trier.de/db/journals/corr/corr1707.html#SchulmanWDRK17

Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. In M. Maher, & J.-F. Puget (Eds.), *Principles and practice of constraint programming — CP98* (pp. 417–431). Berlin, Heidelberg: Springer Berlin Heidelberg.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. Cambridge, MA, USA: A Bradford Book.

Turkeš, R., Sörensen, K., & Hvattum, L. M. (2021). Meta-analysis of metaheuristics: Quantifying the effect of adaptiveness in adaptive large neighborhood search. *European Journal of Operational Research, 292*(2), 423–442. https://doi.org/10.1016/j.ejor.2020.10.045.

Tyasnurita, R., Özcan, E., Shahriar, A., & John, R. (2015). *Improving performance of a hyper-heuristic using a multilayer perceptron for vehicle routing*. Exeter, UK, http://eprints.nottingham.ac.uk/id/eprint/45707