University of Bergen
Department of informatics

Developing an Intelligent Hyperheuristic for Combinatorial Optimization Problems using Deep Reinforcement Learning

Author: Jakob Kallestad
Supervisor: Ahmad Hemmati
Co-supervisor: Ramin Hasibi

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

September, 2021

Abstract

There exist many problem-specific heuristic frameworks for solving combinatorial optimization problems. These can perform well for specific use-cases, however when applied to other problem domains these frameworks often do not generalize well. Metaheuristic frameworks serve as an alternative that aims to be more generalizable to several problems, yet these frameworks can suffer from poor selection of low-level heuristics during the search. The adaptive layer of the metaheuristic framework of Adaptive Large Neighborhood Search (ALNS) is an example of a heuristic selection mechanism that selects low-level heuristics based on their recent performance during the search. In this thesis, we propose a hyper-heuristic selection framework that uses Deep Reinforcement Learning (Deep RL) to more efficiently select heuristics during the search compared to the adaptive layer of ALNS. Our framework uses the representation power of Deep Learning (DL) together with the decision making capability of Deep RL for processing search states (containing useful information about the search) in order to efficiently select heuristics at each step of the search. In this thesis, we introduce Deep Reinforcement Learning Hyperheuristic (DRLH), a general framework for solving combinatorial optimization problems. Our experiments show that DRLH is able to come up with better heuristic selection strategies compared to ALNS and a simple Uniform Random Sampling (URS) framework, resulting in better solutions. Additionally, we show that DRLH is not negatively affected by having a large pool of heuristics to choose from, while ALNS does not perform well under these conditions, as it is unable to work efficiently when given a large pool of heuristics to select from.

Acknowledgements

First and foremost, I would like to thank my supervisors Ahmad Hemmati and Ramin Hasibi for their excellent advising and guidance throughout the master period. Through our many meetings you have provided me with lots of insight and help that has kept this project moving forward since day one. Without the help from both of you this thesis would not have been possible. I would also like to thank my family and friends who helped me stay motivated and take healthy breaks from time to time. Finally, a big thank you to my girlfriend, Anne, for your all your kindness and support. Having people like you in my life has helped me get through this long project.

Jakob Kallestad
01 September, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

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 et al., 2019). Moreover, there is a large body of literature that addresses solving combinatorial optimization problems using metaheuristics. Among these, ALNS (Ropke and Pisinger, 2006) is one of the most widely used metaheuristics. It is a general framework based on the principle of Large Neighborhood 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. According to (Turkeš et al., 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 randomly sampled according to the fixed probabilities of the segment (as seen in Fig. 6.5c). 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 (See Fig. 6.3). 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 and 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.

Reinforcement learning (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 rewards 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 and 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 thesis, we propose **Deep Reinforcement Learning Hyperheuristic (DRLH)**, a general selection hyperheuristic framework (definition in section 2.3) for solving combinatorial optimization problems. In DRLH, we replace the adaptive layer of ALNS with a Deep RL agent trained using Proximal Policy Optimization (PPO) method of (Schulman et al., 2017) responsible for selecting heuristics at each iteration of the search. Our 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. 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 at 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 (see Fig 6.5b). 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.

Our experiments also show that the performance of DRLH is not negatively affected by increasing the number of available heuristics to choose from. In contrast to this, ALNS struggles at handling a high number of heuristics to choose from. This advantage of our framework makes the development process for DRLH very simple as DRLH can automatically discover the effectiveness of different heuristics during the training phase without the need for initial experiments in order to reduce the set of heuristics manually. DRLH is also able to more effectively make use of a wider range of operators for different problem variations and conditions without this negatively affecting its average performance.

1.2 Thesis Outline

The outline for the rest of the thesis is as follows.

Chapter 2 - Background and Related Work gives the theoretical background related to combinatorial optimization and reinforcement learning required for this thesis. It also covers related work on hyperheursitics and previous attempts at solving combinatorial optimization problems using deep reinforcement learning.

Chapter 3 - Problem Sets describes the dynamics of the four combinatorial optimization problems used as example problems that can be solved by DRLH.

Chapter 4 - DRLH introduces the Deep Reinforcement Learning Hyperheuristic model used in this thesis, and provides a detailed look into the framework and also the heuristics used.

Chapter 5 - Experimental Setup contains the specifics of how the experiments were conducted. This includes the hardware used to run the experiments, information about the baseline methods used, hyperparameters and generation of the datasets for each of the problems.

Chapter 6 - Results describes the findings of the experiments and discusses their relevance and significance.

Chapter 7 - Conclusion and Future Work summarizes and concludes the thesis, and finally looks at ideas for future work related to the thesis.

Chapter 2

Background and Related Work

2.1 Combinatorial Optimization Problems

Optimization is the science of making the best possible decision. The main purpose is to get insights into the systems and to find possible solutions for the decision problems (Lundgren et al., 2003). Combinatorial optimization is about finding an optimal object from a finite set of objects (Schrijver, 2003). Combinatorial optimization problems are concerned with the efficient allocation of limited resources that can only be divided into discrete parts in order to optimize an objective. These resources may be machines, people, or other discrete inputs, and the divisibility constraint may restrict the possible alternatives to a finite set. Furthermore, instances of realistic sizes usually have too many alternatives to make complete enumeration a tractable option. For example, a car manufacturer may want to determine the optimal amount of different models to produce in order to maximize profit, an airline may need to set up schedules for the crew such that they minimize the total operating cost, or a flexible manufacturing facility may want to schedule production for a plant without knowing exactly which parts are going to be needed in the future. The difference between make-do planning and those that use sophisticated mathematical models in order to plan the optimal course of actions can be the difference of whether or not a company survives in today's competitive industrial environment (Hoffman and Ralphs, 2013). Applications of combinatorial optimization range to many fields and domains, including: operations research, algorithm theory, applied mathematics, artificial intelligence, machine learning, software engineering and more. Examples of Combinatorial Optimization Problems (COPs) are but not limited to Vehicle Routing Problems (VRP), the Minimum Spanning Tree (MST), Scheduling Problems (SP), Bin Packing Problem (BPP) and the Knapsack Problem (KP). In this thesis we will mainly focus on variations of Vehicle Routing Problems and a Scheduling Problem, but the main contribution of this work, DRLH, is readily applicable to other COPs as well.

Routing Problems

One of the most studied applications of combinatorial optimization problems are Routing Problems that originate from the Traveling Salesman Problem (TSP). TSP is one of the most well-known and fundamental problems in the area of combinatorial Optimization Problems that has been widely researched for decades. In this problem a salesman has to visit a number of cities exactly once and then return to his original starting location. This can be formulated as a NP-Hard optimization problem of trying to find the minimum distance Hamiltonian Cycle.

Among a massive number of routing problems in the literature, the Vehicle Routing Problem (VRP) has received a lot of attention. In VRP multiple routes are conducted from the depot, usually to satisfy some additional constraint of the route such as a maximum weight limit (CVRP) or to stay within specific time windows of the nodes (VRPTW). VRP variants build up a rich family of routing problems.

The Pickup and Delivery Problem (PDP) is a generalization of the classical VRP variants. In PDP the requests consist of a pickup location and a delivery location. In order to complete the request, the vehicle must first go to the pickup location to get the item, and then drop it off at the delivery location. The classical version of PDP is where each request has exactly one pickup location and one corresponding delivery location.

In this work, considering the time frame of a master thesis, we limited ourselves to examine our proposed algorithm on a set of routing problems and also a simple scheduling problem as they are proper representatives for all combinatorial optimization problems. The problems addressed in this thesis will be detailed in chapter 3.

2.2 Solution Methods

There are many approaches for solving combinatorial optimization problems. For a better overview we can divide these approaches into two main categories, known as *exact approaches* and *heuristic approaches*. We will primarily look into the heuristic approaches and it's many sub-variants as this is the main focus of this thesis, but before that we will briefly explain what constitutes an exact approach in order to provide some context on alternative methods for solving COPs.

2.2.1 Exact Approach

An exact approach for an optimization problem will always find a global optimal solution, but can often take longer to run than a heuristic approach. This is especially true for real world problems with many constraints and large instance sizes. In order to guarantee optimality, exact approaches need to explore a sizable portion of the solutions in the solution space. Examples of exact approaches include branch-and-cut, branch-and-bound, branch-and-price (Costa et al., 2019).

2.2.2 Heuristic Approach

Heuristic approaches are able to find good solutions in a short amount of time. The solutions found are not required to be optimal, but are usually satisfactory to the specific use-case of the search. Heuristic approaches are typically much faster than exact approaches and may be the only alternative on large instance sizes where finding the global optimal best is not tractable with exact approaches. For this reason heuristics are very popular and have been widely studied by the optimization community for several decades. Two main ways of conducting a search using a heuristic approach can be divided into constructive heuristics which seeks to sequentially build up a solution from scratch, and perturbative heuristics (also called local search heuristics) which seeks to modify an existing solution in order to improve it. We will go into more details for each of these in the following.

Constructive Heuristics

A constructive heuristic is one that builds up a solution one element at a time until a finished solution has been constructed. These approaches are generally able to find much better solutions than random approaches, but may struggle to find anything close to that of exact approaches or perturbative heuristic approaches. An advantage of constructive heuristics is that they can usually find solutions very quickly, and because of this they are often used in order to build an initial feasible solution for perturbative approaches which is later improved. An example of a constructive heuristic for the TSP is the *Nearest Neighbor* heuristic which visits the closest node at each step until all the nodes have been visited.

Perturbative Heuristics

A perturbative heuristic is one that modifies an existing solution in order to create a new solution. This is the most common type of heuristics in optimization research. Such heuristics can be adjusted to a specific problem or they can be general heuristics applicable to multiple problems, such as swap or exchange. It is normal to combine many perturbative heuristics into a pool of heuristics as this has shown to give better results than using only a single heuristic throughout the entire search (Fisher and Thompson, 1963). An important aspect of perturbative heuristics is how much they should change the solution. This aspect is addressed by the concept of a *neighborhood.* The neighborhood of a solution is defined as all the possible solutions that can result from using one of the heuristics on the solution. In order to avoid getting stuck in a local optima, it can be beneficial to have a large neighborhood. A large neighborhood can be achieved by having a good mix of *diversification* and *intensification* within the heuristics. Diversification helps change the solution, even if it results in a worse objective value, in order to escape local optima. On the other hand, intensification helps to find solutions that strictly improve the objective value of the solution. Examples of frameworks that employ large neighborhoods are the Large Neighborhood Search (LNS) framework of (Shaw, 1998), and later the ALNS framework of (Pisinger and Ropke, 2007). These methods work by applying destroy and repair steps repeatedly on the solution which provides a good balance of intensification and diversification.

2.2.3 Metaheuristics

A metaheuristic is a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms (Sörensen and Glover, 2013).

The field of metaheuristics aims to search within a search space of problem solutions. They make few assumptions about the optimization problems being solved and can therefore be seen as problem-independent approaches, meaning that they can be applied to a wide range of different problems. The most widely used metaheuristics are designed for perturbative heuristics and includes the likes of: Simulated Annealing (SA), Tabu Search (TS), Adaptive Large Neighborhood Search (ALNS), Genetic Algorithms (GA). There also exist construction-based metaheuristics such as Sampling and Beam Search (BS) which lets constructive heuristics find much better results than they would have been capable of finding without a metaheuristic guiding the search.

2.3 Hyperheuristics

A hyperheuristic is a heuristic search method that seeks to guide the selection or generation process of heuristics in order to more efficiently solve combinatorial optimization problems. The term *hyperheuristic* was first used in the context of combinatorial optimization by Cowling et al. (2001) and described as *heuristics to choose heuristics*. Burke et al. (2010b) later extended the definition of hyperheuristic to 'a search method or learning mechanism for selecting or generating heuristics to solve computational search problems'.

Burke et al. (2010a) classified hyperheuristics according to two dimensions (i) the nature of the heuristic search space, and (ii) the source of feedback during learning. (i) creates a distinction between *selection hyperheuristics* and *generation hyperheuristic*, and further classification can be made to differentiate between *construction* and *perturbation* hyperheuristics. (ii) separates the sources of feedback during learning such as *Online*, *Offline* and *No Learning*. This thesis will focus on *selection hyperheuristics* which is concerned with selecting heuristics during the search, and we will be comparing methods that use different sources of feedback in the result section.

An important aspect of hyperheuristics is that there should exist a domain barrier between the dynamics of the underlying problem and the mechanisms for selecting or generating heuristics. In other words, hyperheuristic research has long focused on creating problem-independent approaches applicable to a wide range of problems in favor of specifically designed approaches that might obtain better results for specific problems. Later in this thesis we will present DRLH, a general selection hyperheuristic using deep reinforcement learning, which we will compare to two simpler selection mechanisms, namely ALNS and URS.

2.4 Reinforcement Learning

Machine learning is often divided into three fields: supervised learning, unsupervised learning and reinforcement learning. In supervised learning there exists a training set of input-output pairs where the goal is to learn a behavior that most closely resembles that of the input-output pairs found in new unseen data from a similar distribution as the training set. Unsupervised learning also concerns a dataset, but the tasks usually consist of exploring the underlying structure of the data itself and can be used to discover clusters, anomalies or other interesting properties of the data. This thesis will focus on the field of RL, which is different from both supervised and unsupervised learning in that a dataset is not provided. Instead RL learns by interacting with an environment in which each action results in a reward and the goal is to earn as much reward as possible during an episode consisting of a number of steps. The types of problems that RL tackles are those in which the underlying model of the environment is unknown and is affected by the agent's choice of actions (Sutton and Barto, 2018). Applications of RL takes place in a wide range of fields including: games (Mnih et al., 2015, Silver et al., 2016), robotics (Kober et al., 2013, Levine et al., 2016, 2018), vehicle routing problems (Kool et al., 2019, Nazari et al., 2018), autonomous driving (Sallab et al., 2017), drug design (Popova et al., 2018), health and medicine (Frank et al., 2005, Zhao et al., 2009), Media and Advertising (Abe et al., 2004, Agarwal et al., 2016, Cai et al., 2017), Finance (Bertoluzzo and Corazza, 2014), text, speech and dialog systems (Dhingra et al., 2017, Paulus et al., 2017) and others.

In this chapter we will give an introduction to RL and its core concepts. Then we will look at deep reinforcement learning and see how integrating deep learning into RL enables even more powerful approaches such as policy gradient methods.

2.4.1 Introduction to Reinforcement Learning

Reinforcement learning (RL) is a *computational* approach to learning with the goal of taking a sequence of actions in order to maximize a numerical reward without manually specifying which actions to be taken. RL consists of an *agent* whose task is to take a sequence of actions in a dynamic *environment*. The environment provides the agent with observable *states* that the agent can use to decide which actions to take and *rewards* that inform the agent about how desirable it is to take actions in the corresponding states.

This can be written more formally as, an agent observes a state s_t from the environment at step t. The agent then decides to take action a_t, changing the environment so that it transitions to a new observable state s_{t+1} and also provides a reward r_{t+1} as feedback to the agent. The goal of the agent is to maximize the cumulative reward of an *episode* (consisting of a number of steps), and doing so requires the agent to follow a *policy* π, serving as a strategic plan for what actions to take in any given state of the environment. To achieve this, the learner has to find out which actions result in the higher reward through a trial and error process of interacting with the environment and observing the outcome of actions taken in specific states. In the case of more complex tasks the effect of an action might be more than just the immediate reward as an action can also affect the rewards received in subsequent steps.

A *policy* determines the behavior of the agent. The probability of taking action $a_t \in \mathcal{A}(s_t)$ in state $s_t \in \mathcal{S}$ is modeled as the probability distribution $\pi(a_t|s_t)$, and the agent uses this to sample its next action from $\pi(a_t|s_t)$. It works as a mapping between the states received from the environment to the actions that should be selected by the agent for those states. The policy is the most essential part of the learning agent, as it is sufficient to understand the behavior of the agent. In terms of what constitutes a policy, it can range from a simple function or lookup table to more complex systems such as neural networks.

The *reward function* essentially defines the goal of the task at hand in a numerical way that can be optimized by RL. Rewards can be dense or sparse, meaning that they can either be given as feedback to the agent at every step of an episode or only at specific steps such as the final step of the episode. The goal of the agent is to maximize the cumulative reward received during an episode. Because each action affects the environment in some way it is not always advantageous for the agent to take the action giving the highest immediate reward, as the agent should also take into account the total future reward it can receive and try to take the action that maximizes this. Due to the way the reward function influences the intended objective of the agent, it is the main source of change to the policy of an agent, and greatly affects the effectiveness of the learning process.

A *value function* $v(s)$ indicates how good it is to be in a state. While the reward function only gives the immediate return, the value function estimates the expected future reward received from the current state until the end of the episode. The reward function thus indicates the immediate desirability of the state, while the value function gives an approximation of the long-term benefit of the state by also accounting for the rewards that are likely to follow in future steps from the given state.

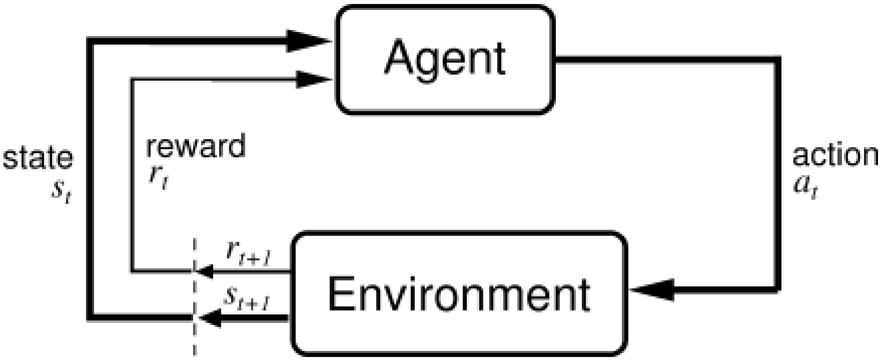


Figure 2.1: At each step $t = 1, 2, 3, ...$ the agent receives an observation (state) $s_t \in \mathcal{S}$ from the environment and takes one of the available actions $a_t \in \mathcal{A}(s)$. This action changes the environment in some way, and a new observation s_{t+1} and a reward r_{t+1} is given to the agent. Figure from (Sutton and Barto, 2018).

One way to categorize RL approaches is to make the distinction between: value-based methods, policy-based methods and actor-critic methods (Sutton and Barto, 2018). The distinguishing factors between these approaches are that:

- Value-based: Has an implicit policy based on a learnt value function.
- Policy-based: Does not have a value function. Uses a learnt policy function to select actions.
- Actor-critic: Has both a learnt policy and a learnt value function.

Value-based methods use a value function to estimate the expected return of being in a state. One commonly used value function is the *state value function* $V^{\pi}(s) = \mathbb{E}(R|S)$

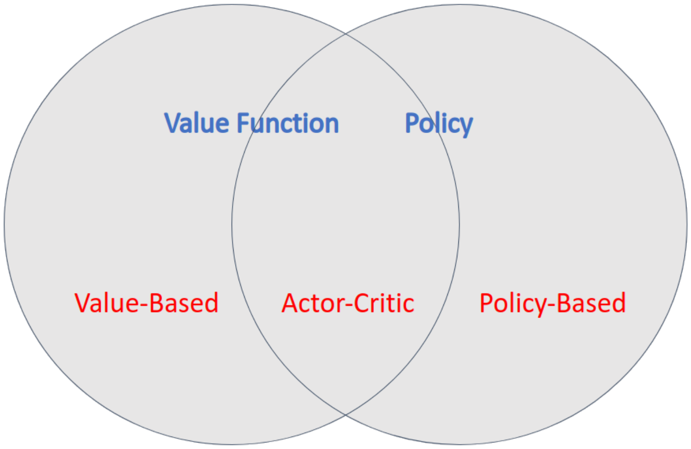


Figure 2.2: Venn diagram illustrating the relationship between value-based, policy-based and actor-critic methods.

which estimates the expected return based only on the input state. Another value function is the quality function (Q-value) $Q^{\pi}(s,a) = \mathbb{E}(R|s,a)$ that uses both the initial action a and the initial state s in order to estimate the expected return. Using $Q^{\pi}(s,a)$ the agent can greedily choose the a that maximizes the quality function at each step. Finding $Q^{\pi}(s,a)$ can therefore be used to effectively solve RL problems. The most common way to estimate the quality function relies on the recursive definition of the Bellman equation (Bellman, 1954).

$$Q^{\pi}(s,a) = E_{s_{t+1}}[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi(s_{t+1}))]$$

Policy-based methods are different from value-based methods in that they learn an optimal policy directly without using a value function. Such policy in these methods is typically parametrized as a vector θ with π_{θ} acting as a stochastic function parametrized by θ that takes states and actions as inputs, and outputs a probability distribution over the available actions. A common occurrence in policy-based methods is to use the weights of a neural network to correspond to the parameter θ, which we will cover in detail in the next section. Policy-based methods typically suffer from a high variance, although some measures can be taken in an attempt to reduce the variance, such as comparing the performance of the agent to a baseline, e.g., random action selection.

Actor critic methods combine the benefits from using a value function and a policy in order to improve learning. The actor (policy) is responsible for selecting actions to perform, while the critic (value function) attempts to estimate the value of the states so that the actor can maximize the expected return. The inclusion of value functions in policy-based methods results in a compromise between high bias and high variance (Konda and Tsitsiklis, 2001, Schulman et al., 2018). Several actor critic methods have been shown to be state-of-the-art approaches when used in a policy gradient setting, which we will cover in detail in the next section.

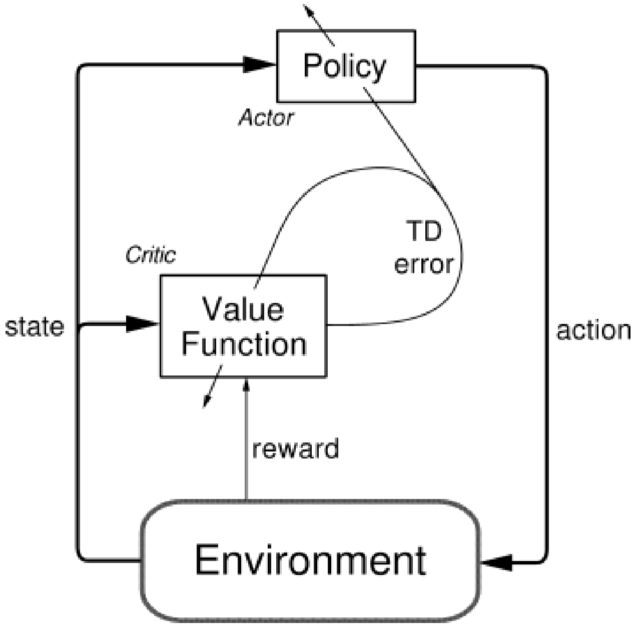


Figure 2.3: Actor-critic architecture. The policy is represented by the actor and maps input states to output actions. The value function is represented by the critic. During the learning phase the both the actor network and the critic network is updated in which the critics evaluation of states contributes to the update of the actor. Figure from (Sutton and Barto, 2018).

2.4.2 Deep Reinforcement Learning

Deep reinforcement learning is a combination of RL and deep learning which allows agents to work on unstructured input data through the use of neural networks, omitting the need to manually engineer the state space. Neural networks provide excellent representation learning properties that can create rich and complex models which can be used to tackle complex problems in a variety of fields such as object detection (Baird and Wang, 1995), speech recognition (Bengio et al., 2013) and language translation (Vaswani et al., 2017). The representation learning properties of neural networks makes them able to learn from high dimensional input data as well as scale effectively to high dimensional problems. This makes them an ideal fit to be used in RL in which classical RL methods suffer from the explosion of the dimensionality of the state and action spaces for complex problems. The idea is that neural networks can be used to approximate the optimal policy π^*, value function $V*$ and Q-value $Q*$.

While classical RL approaches rely on tabular representation which limits the number of states and actions that it can handle, the inclusion of deep neural networks as a function approximator can be used to overcome these restrictions. This is useful as the state space of real world problems is often very large, making it infeasible to explore all of the states and action-state pairs. Additionally, large state spaces might not be representable in tabular approaches due to hardware limitations as the size of the table needs to fit in memory for it to work. Another benefit of using neural networks as a function approximator is that relevant features from raw inputs can be generalized across similar states, leading to better representation and lower training times. This also makes the agent able to work well when faced with unseen states, which is not the case when using tabular methods.

Value function methods can benefit from deep learning by replacing the one-to-one value table with a deep neural network as a function approximator. The output of the network is used as probabilities for each of the possible actions, and is typically sampled or greedily selected by the agent. A popular value function method using deep RL was introduced by Mnih et al. (2015) called Deep Q-Network (DQN). This method combines Q-learning with neural networks and has been shown to be highly effective when applied to a wide range of tasks. This approach later led to a surge of similar methods seeking to improve on DQN, such as Double DQN (van Hasselt et al., 2015) and Rainbow DQN (Hessel et al., 2017).

Policy gradient methods are a subclass of policy search methods from classical RL in which the policy $\pi(a|s, \theta)$ parameter θ is encoded by the weights of a neural network. Consequently, we observe that optimizing the parameters of θ to find the optimal policy is equivalent to optimizing the weights of the neural network. This is beneficial as deep learning techniques such as back-propagation can be used in order to find the optimal policy, and because of this network-based deep RL methods have been very successful for solving RL tasks. An advantage of policy gradient methods is that they often provide a stable convergence property which improves smoothly at each time step. This is in contrast to value-based methods, where updating the value function can often cause drastic changes to the behavior, leading to large oscillations when training. Another advantage of policy gradient methods is that they work well in the face of uncertainty as they are able to effectively learn stochastic policies. A disadvantage of policy gradient methods is their tendency to converge to a local optima instead of the global optima (Sutton and Barto, 2018).

Proximal Policy Optimization (PPO) (Schulman et al., 2017) is a state-of-the-art policy gradient method that has gained a lot of popularity due to its relative speed and ease of use compared to other state-of-the-art methods. It is based on the actor-critic architecture and is able to handle both discrete and continuous action spaces. The main contributions of PPO are (1) the Clipped Surrogate Objective and (2) the use of multiple epochs of stochastic gradient ascent to perform each policy update. The Clipped Surrogate Objective is designed to improve training stability by limiting the change made to the policy at each step, thus ensuring that training is less likely to diverge. Furthermore, in contrast to vanilla policy gradient methods, PPO allows running multiple epochs of gradient ascent on the sampled training data without causing destructively large policy updates. This allows it to squeeze more out of the training data and reduce sample inefficiency. The combination of both high data efficiency and reliable performance makes PPO an excellent first choice when attempting to solve a problem using deep RL. PPO has been applied to solve many problems, such as improving the state-of-the-art performance on Atari games and Mujoco robotic tasks. PPO is also the method that we have elected to use as the backbone in our implementation of DRLH.

2.5 Related Work

Özcan et al. (2010) propose a hyperheuristic method that uses a RL hyperheuristic 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 type of 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.

A train and test hyperheuristic method for the Vehicle Routing Problem named *Apprentice Learning-based Hyper-heuristic (ALHH)* was proposed by Asta and Özcan (2014) in which an apprentice agent seeks to imitate the behavior of an expert agent. 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 et al. (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 and behavior. 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 considers 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 are richer in that they contain more information compared to the search state of the other two methods.

In addition to hyperheuristic approaches there have also recently been many attempts at solving popular routing problems using Deep RL by the ML 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 et al., 2019, Nazari et al., 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 in 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 and Tierney (2019) have used a Deep RL agent to generate a heuristic for rebuilding partially destroyed routes in the Capacitated Vehicle Routing Problem (CVRP) in 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 and 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 as well as the input and type of ANNs used for the agents from one problem to another changes 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 et al. (2020) suggested the use of a Deep RL agent for choosing low-level heuristic at each step for the CVRP problem. This work also suffers from the generalizability to other types of optimization problems due to the elements of Deep RL agent that is 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 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.

Chapter 3

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).

3.1 Capacitated Vehicle Routing Problem (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 available vehicles. Additionally, there is a depot in which the vehicles travel from and return to when serving the orders. 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.

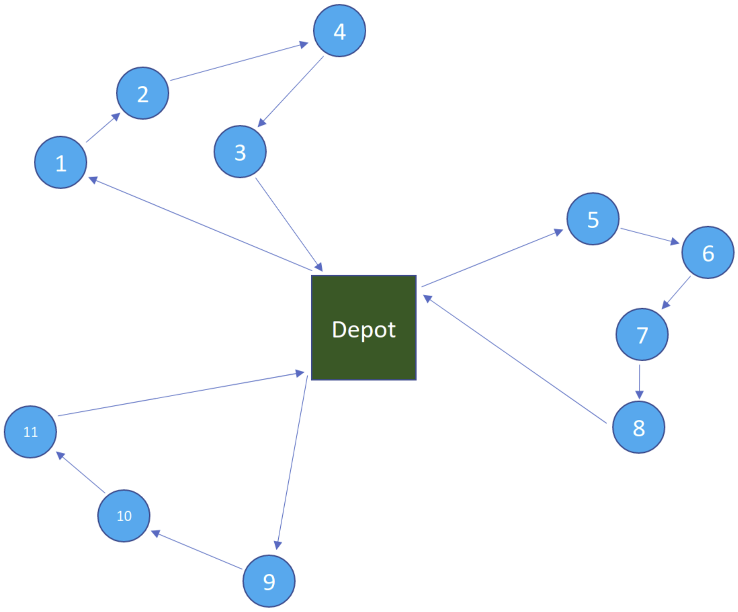


Figure 3.1: CVRP Problem Illustration

3.2 Parallel Job Scheduling Problem (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 then there is a delay. 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.

3.3 Pickup and Delivery Problem (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

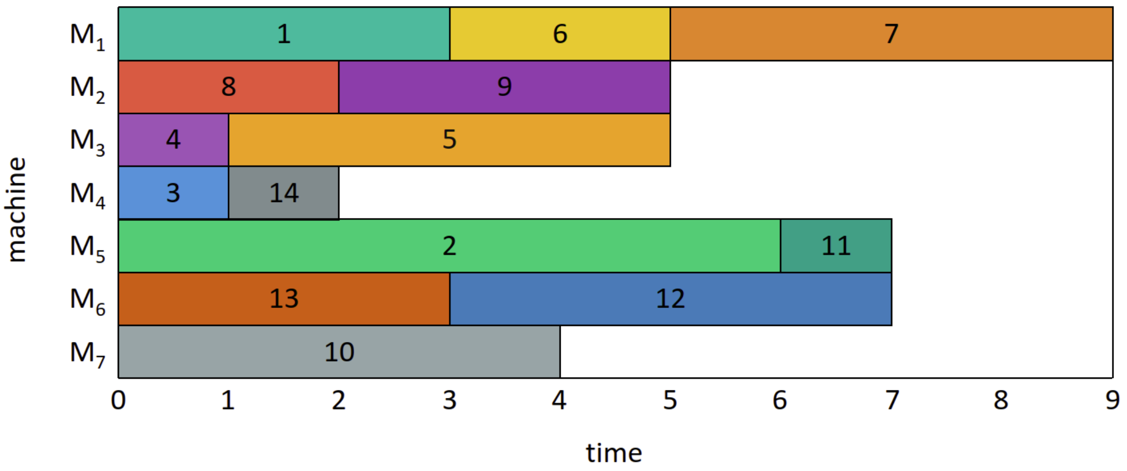


Figure 3.2: PJSP Problem Illustration

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.

3.4 Pickup and Delivery Problem with Time Windows (PDPTW)

In the Pickup and Delivery Problem with Time Windows, 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.

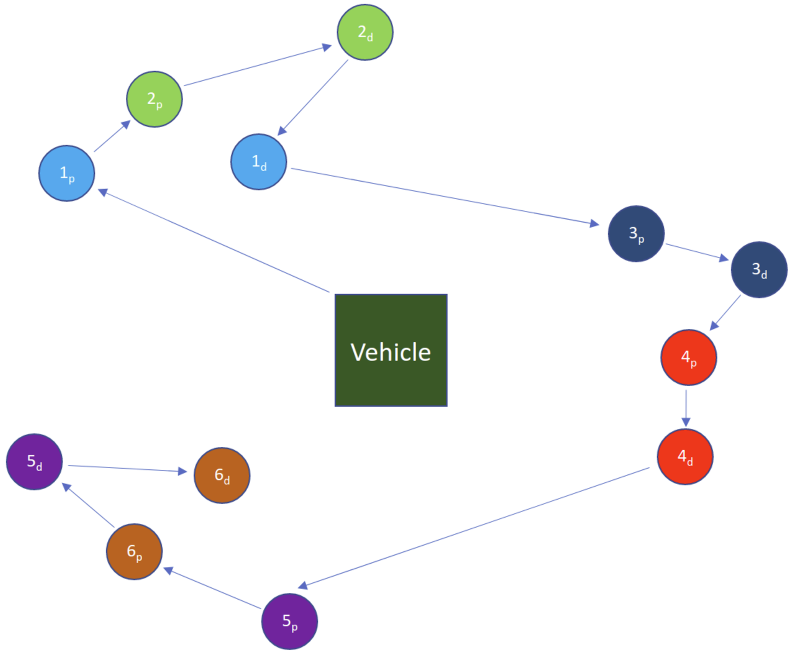


Figure 3.3: PDP Problem Illustration

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 et al., 2014).

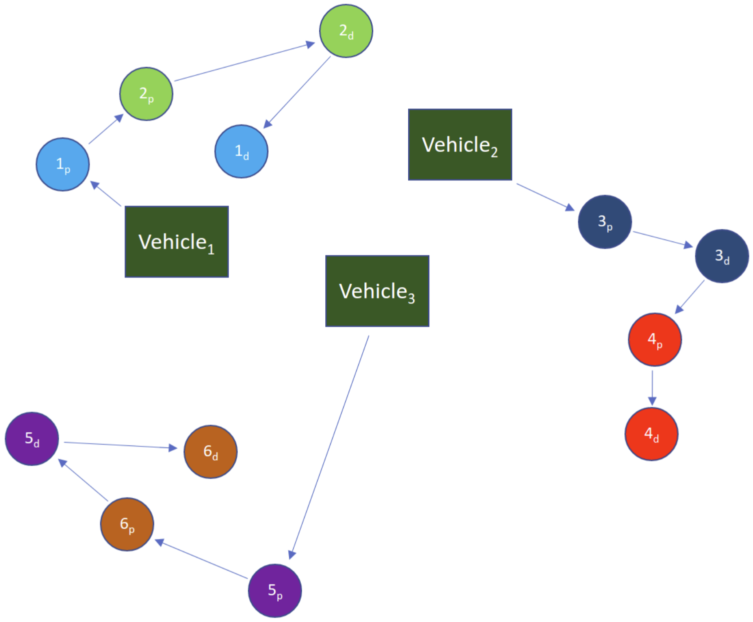


Figure 3.4: PDPTW Problem Illustration

Chapter 4

DRLH

In this chapter, we present the DRLH, a hyperheuristic approach based on Deep RL and ALNS. We illustrate the performance of this novel hyperheuristic by applying it to the different problems discussed in chapter 3. Our proposed hyperheuristic framework uses an RL agent for the selection of heuristics. This process improves on the ALNS framework of (Ropke and 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

4.1 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 l 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 problems in chapter 3.

Algorithm 1: DRLH

```
Function Deep Reinforcement Learning Hyperheuristic
    Generate an initial solution $l$ with objective function of $f(l)$ (see section 4.5)
    $H$=Generate_heuristics() (see section 4.1)
    $l_{best} = l, f(l_{best}) = f(l)$
    Repeat
        $l^{'} = l$
        choose $h \in H$ based on policy $\pi(h|s, \theta)$ (see section 4.4)
        Apply heuristic $h$ to $l^{'}$
        if $f(l^{'}) < f(l_{best})$ then
            $l_{best} = l$
        end
        if $accept(l^{'}, l)$ (see section 4.3), then
            $l = l^{'}$
        end
    Until stop-criterion met (see section 4.3)
    return $l_{best}$
```

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 $H$
```

4.2 Sample Set of Heuristics

Each heuristic $h \in H$ is a combination of a removal and an insertion operator presented in Tables 4.1 and 4.2. Furthermore, one additional intensifying heuristic is also added to H. In each iteration, a heuristic $h \in H$ is applied on the incumbent solution l with cost of $f(l)$ and generates a new solution $l^{'}$ with cost of $f(l^{'})$. For our sample set of heuristics, H has the size of $|H| = 29$ (7 removals × 4 insertions + 1 additional).

4.2.1 Removal Operators $\mathcal{R}$

The set of all removal operators $\mathcal{R}$ are provided in Table 4.1. 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.

Table 4.1: 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

4.2.2 Insertion Operators $\mathcal{I}$

Table 4.2 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 l'. Operator "*Insert_greedy*" places each removed element in the position which obtains the minimum total cost of the new solution $f(l')$. Operator "*Insert_beam_search*" performs beam search with search depth of 10 for placing each removed element. Beam search keeps track of 10 the best position combinations after inserting each removed element in the solution and puts the elements in the best combination of positions that obtain the minimum $f(l')$ 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.

Table 4.2: 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

4.2.3 Additional Heuristics $\mathcal{C}$

Unlike in ALNS where only removal and insertion operators are used, our framework can also make use of standalone heuristics that employ 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 l for the element that achieves the minimum cost $f(l')$. "*Find_single_best*" is the only additional heuristic that is used in the proposed sample set of heuristics, H.

4.3 Acceptance Criteria and Stopping Condition

We use the acceptance criterion $accept(l', l)$ used in simulated annealing (Kirkpatrick et al., 1983). This acceptance criterion depends on the difference in objective value between the incumbent l and the new solution l' denoted as $\Delta E = f(l') - f(l)$ 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:

$$T_0 = \frac{\overline{\Delta E}}{\ln 0.8} \tag{4.1}$$

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

4.4 Deep Reinforcement Learning Agent for Heuristic Selection

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

$$\pi(a|s) = Pr\{A_t = a|S_t = s\}. \tag{4.2}$$

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.(4.2) is redefined as

$$\pi(a|s, \theta) = Pr\{A_t = a|S_t = s, \theta_t = \theta\}. \tag{4.3}$$

in which θ_t represents the parameters at time step t (Sutton and Barto, 2018). In our setting, the policy π is a MultiLayer perceptron (MLP), which is a class of non-linear function approximation (Goodfellow et al., 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 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.

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
```

As mentioned above, three main properties of the RL agent which is 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 makes 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 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.

4.4.1 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 4.3 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

Table 4.3: A list of all features used for the state representation

Name	Description
reduced_cost	The difference of cost between previous & current solutions
cost_from_min	The difference of cost between current & 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 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.

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 (see section 6.7). *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 of 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 (see Table 6.4 for results related to this).

4.4.2 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.(4.3) is redefined as

$$\pi(h|s, \theta) = Pr\{A_t = h|S_t = s, \theta_t = \theta\}. \tag{4.4}$$

4.4.3 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 function that has the above property. We refer to this as R_t^{5310}, the formula for which is

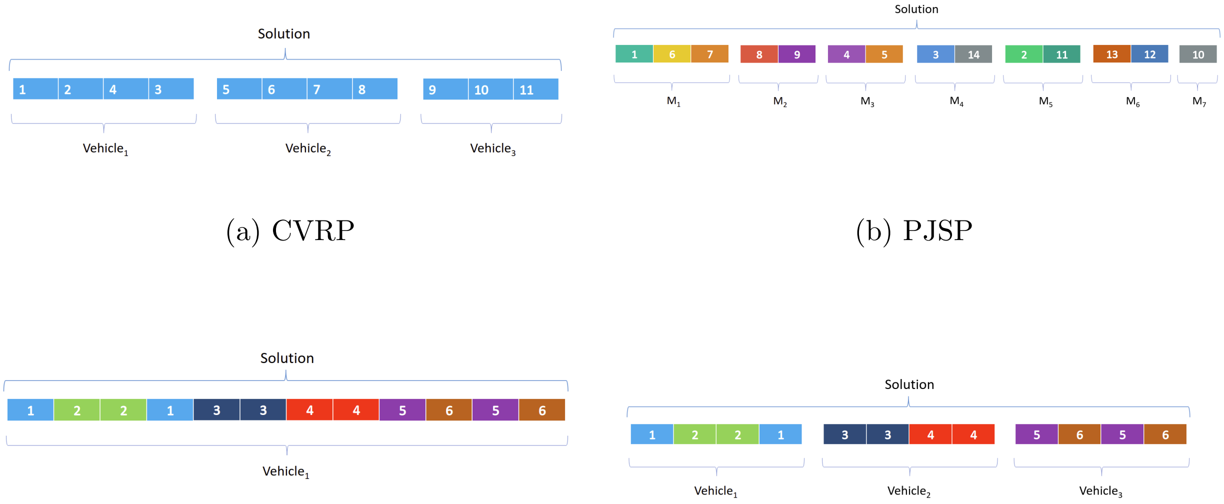
$$R_t^{5310} = \begin{cases} 5, & \text{if } f(l^{'}) < f(l_{best}) \\ 3, & \text{if } f(l^{'}) < f(l) \\ 1, & \text{if } accept(l^{'}, l) \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

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. Experiments on these reward functions demonstrated 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 and Pisinger, 2006), we use the same function for our Deep RL agent and ALNS for an equal comparison.

4.5 Solution Representation and Initial Solution

For all the problems 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 representations 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 4.2. For each of the problems and each test instance, DRLH, ALNS and URS start with the same initial solution for a fair comparison.



(a) CVRP

(b) PJSP

(c) PDP

(d) PDPTW

Figure 4.1: Solution representations for CVRP, PJSP, PDP and PDPTW.

Chapter 5

Experimental Setup

5.1 Experimental Environment

The computational experiments in this thesis 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

Two baseline frameworks are chosen to compare with DRLH. Both of these use the same set of heuristics as DRLH with the difference being how the heuristics are selected. 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 Sampling (URS)

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

5.3 Hyperparameter Selection

The hyperparameters for DRLH 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 DRLH for the experiments are listed in Table 5.1.

In order to decide on the hyperparameters for DRLH we did some initial experiments on a separate validation set to see which combinations performed best. After that we have used the same hyperparameters for all experiments in this thesis and found that they 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.

Table 5.1: The hyperparameters used during training for the Deep RL agent of DRLH

Hyperparameter	Value
Max epochs	5000
Learning rate	1e−5
Batch size	64
First hidden layer size	256
Second hidden layer size	256
Discount factor	0.5

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. The exception is for PDPTW where we utilize a known set of benchmark instances for testing.

5.4.1 CVRP

CVRP data instances are generated in accordance with the generation scheme of (Kool et al., 2019, Nazari et al., 2018), 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 $\{1, ..., 9\}$, 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 $\{100, 101, ..., 1000\}$, 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 (Kool et al., 2019, Nazari et al., 2018). 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 $\{1, ..., 9\}$, 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 5.2. For testing we use the benchmark instances of (Hemmati et al., 2014). This consists of 5 instances of each variation. Previous work by Homsi et al. (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 don't have the global optimal objectives, but instead use the best known values found by DRLH with 10000 iterations to calculate the *Min Gap (%)* and *Avg Gap (%)*. For generating the training set we use the provided instance generator of (Hemmati et al., 2014).

Table 5.2: 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

Chapter 6

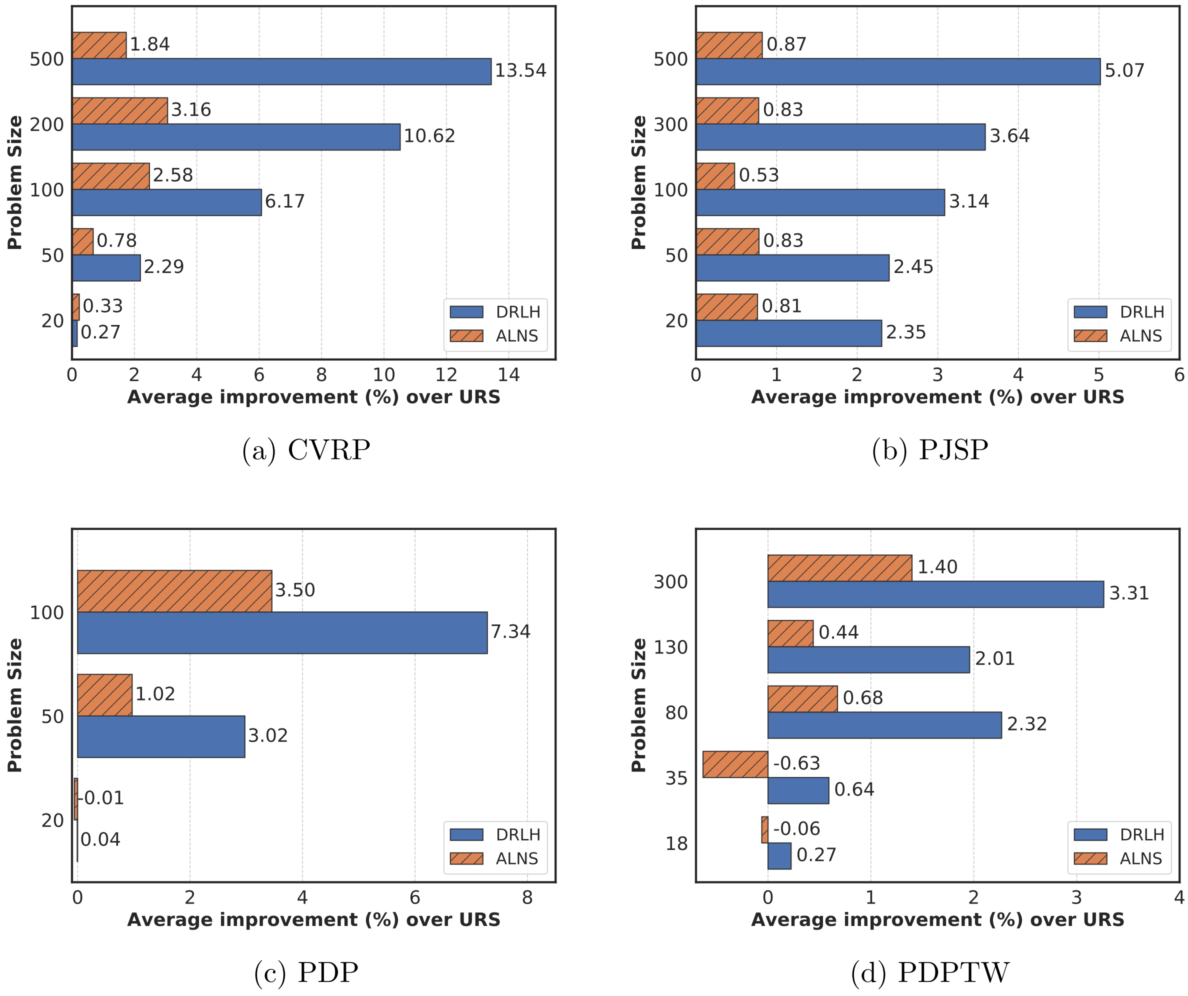
Results

6.1 Results of CVRP

Fig. 6.1a shows the improvement in percentage that using DRLH and ALNS has over using URS on CVRP instances of different sizes. We see that DRLH is able to outperform both URS and ALNS 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 URS and ALNS on larger instance sizes.

6.2 Results of PJSP

Fig. 6.1b shows a similar result as for the CVRP 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.



(a) CVRP

(b) PJSP

(c) PDP

(d) PDPTW

Figure 6.1: Results of DRLH for CVRP, PJSP, PDP and PDPTW after 1000 iterations.

6.3 Results of PDP

Similarly to the other problems we see from Fig. 6.1c that DRLH outperforms ALNS and URS on all the instance sizes tested and that the effectiveness of DRLH seems to increase with larger instance sizes. Fig. 6.2 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, 10000 and 25000 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

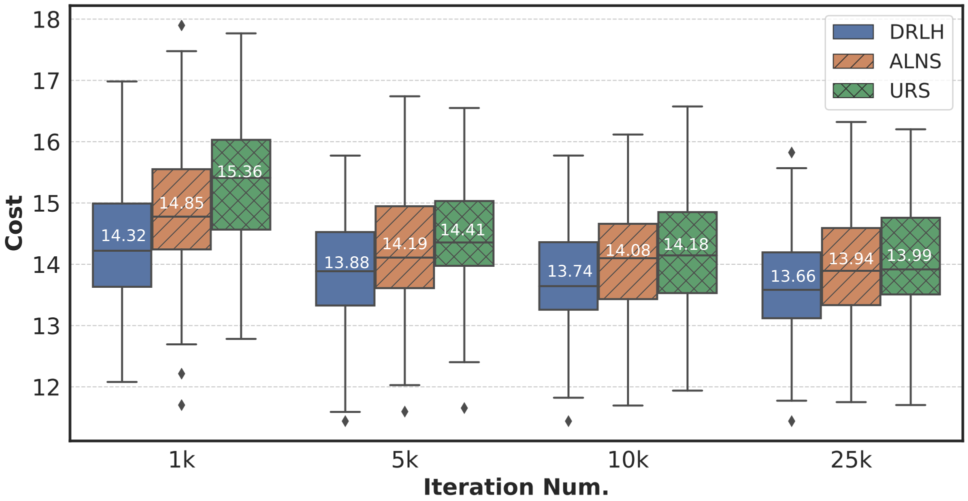


Figure 6.2: Boxplot results for different number of iterations for PDP100

during the search makes the choices of which heuristics to select less sensitive compared to searching for a smaller number of iterations.

6.4 Results of PDPTW

Finally, we observe a similar trend for PDPTW as for the other problems, which can be seen in Fig. 6.1d. From this figure, showing results after 1000 iterations, we see that DRLH outperforms ALNS and URS on all instance sizes tested and that performance tends to increase on larger instance sizes.

In addition to this figure we also report results for PDPTW shown in Tables 6.1, 6.2 and 6.3 for 1000, 5000 and 10000 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 a different strategy for heuristic selection.

Table 6.1: 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

Table 6.2: 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 6.3: Average results for PDPTW instances with mixed call sizes after 10000 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

6.5 Results of an Increased Pool of Heuristics

In addition to the set of heuristics mentioned in section 4.1 we have also created an extended set of heuristics (see Appendix B). In total this extended set consists of 142 heuristics. Fig.

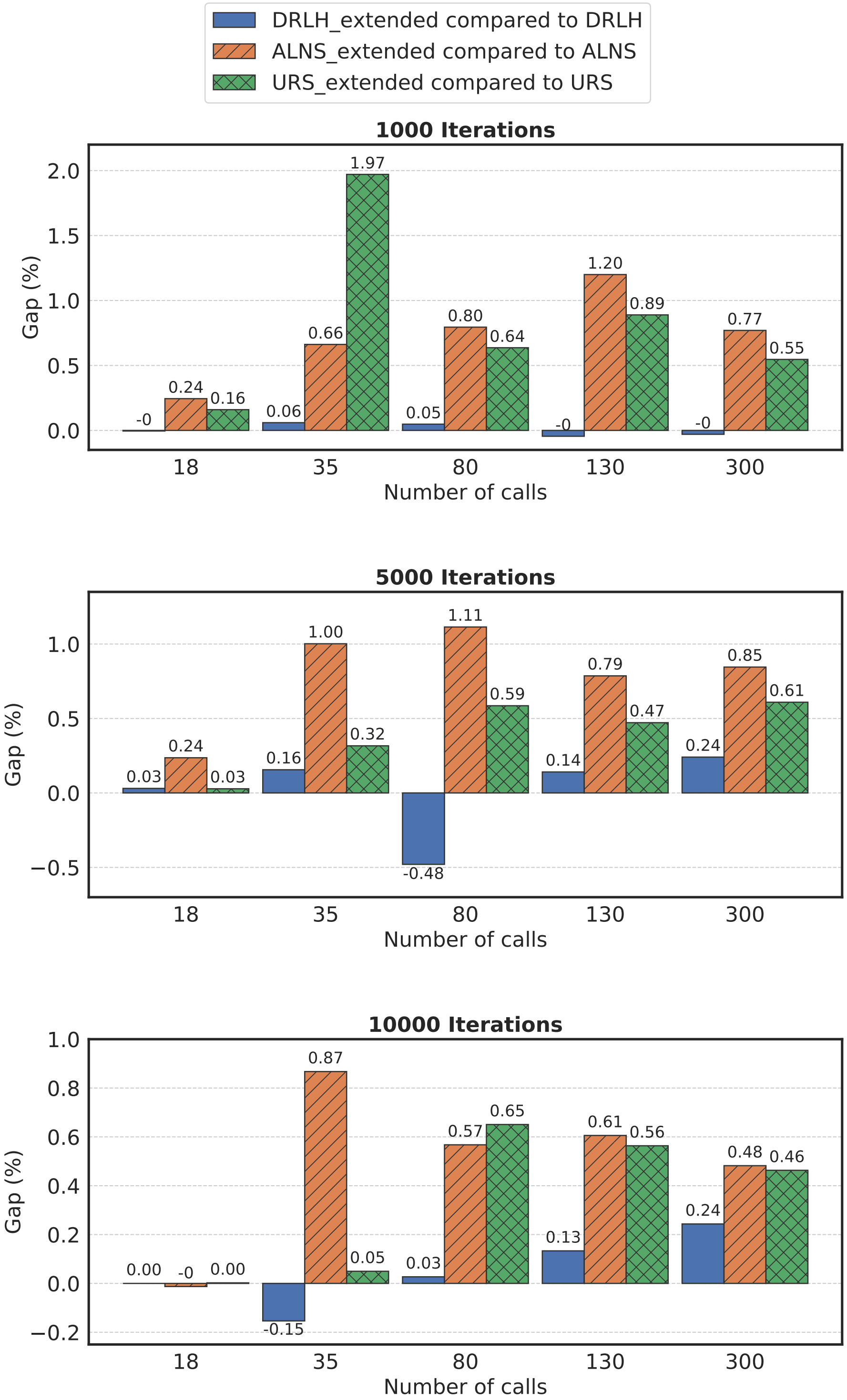


Figure 6.3: Results of an Increased Pool of Heuristics

6.3 shows the average gap of using the extended set compared to using the standard set for each of DRLH, ALNS and URS. 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. 6.3 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 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 this 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.

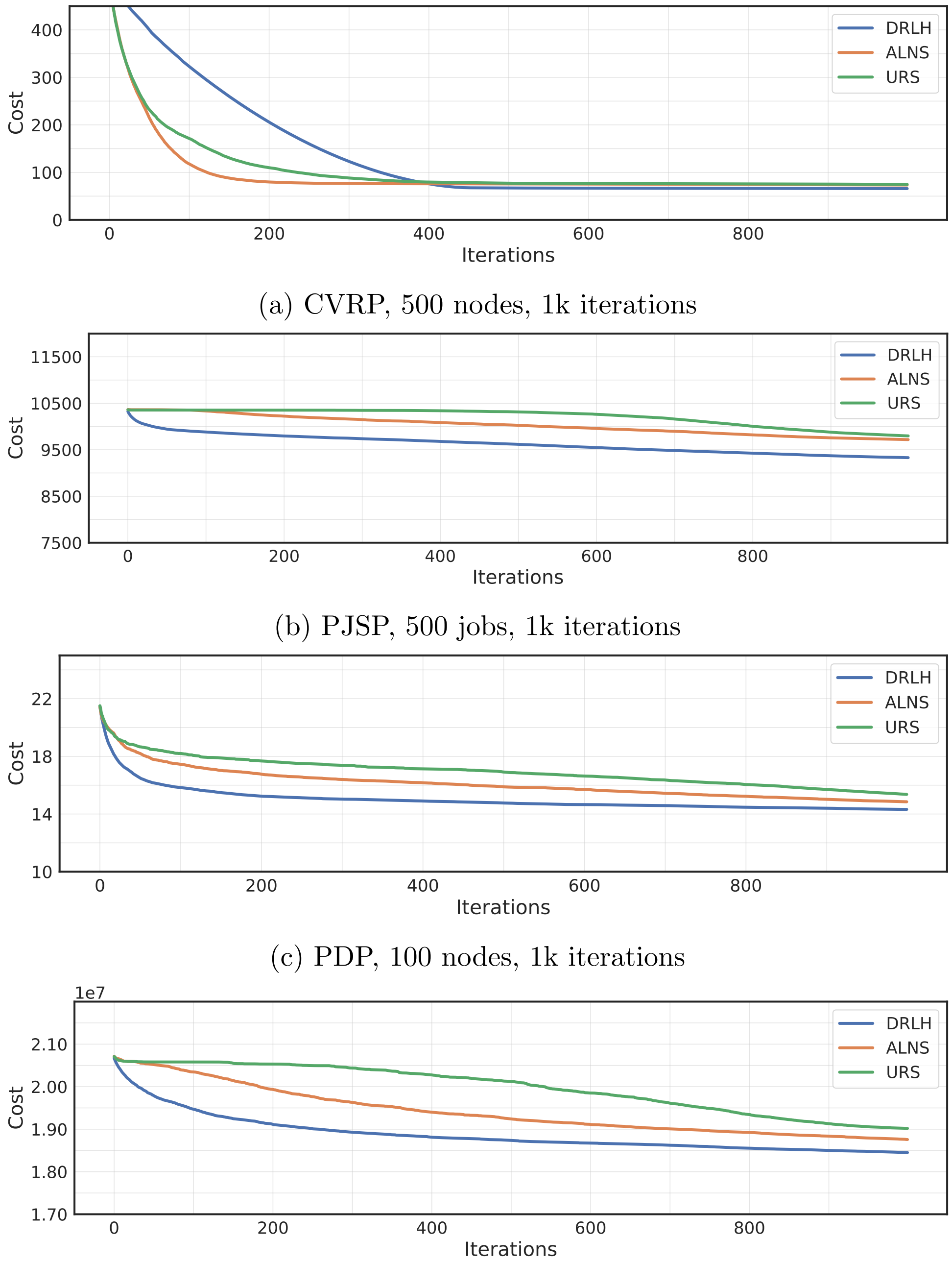
In addition to DRLH having a simpler development phase, an increased (or more nuanced) set of heuristics also has a larger potential to work well for a wide range of conditions, such as for different problems, instance sizes and specific situations encountered in the search. In other words, reducing 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.

6.6 Performance Results

In this section we explore the speed and characteristics of the performance of DRLH, ALNS and URS on the different problems. Fig. 6.4 shows that DRLH is able to quickly find better solutions compared to ALNS and URS for all the problems. Although for CVRP, it 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 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 about 500 iterations to match. We refer to Appendix C for a complete collection of performance plots for all the problems that we have tested.

6.7 Heuristic Selection Strategies

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. 6.5a and 6.5b. 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 making of ALNS within a single segment is random according to the locked probabilities of that segment, see Fig. 6.5c. Depending on the problem and available heuristics to select, there might exist exploitable strategies and patterns for heuristic selection, such as heuristics 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 designed algorithms designed by human experts. One such exploitable strategy found by DRLH on our problem with our provided set of heuristics we will refer to 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.



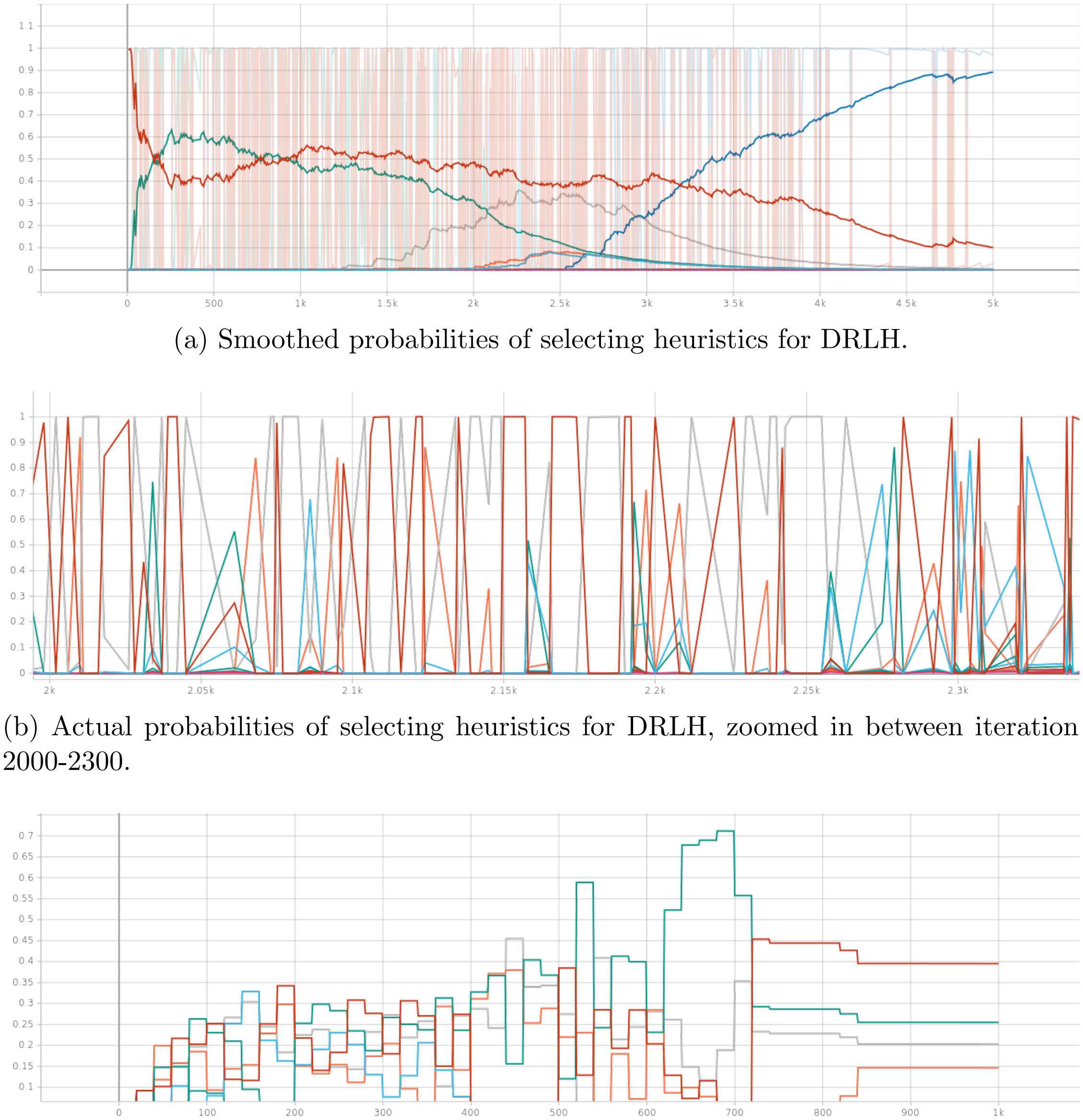
(a) CVRP, 500 nodes, 1k iterations

(b) PJSP, 500 jobs, 1k iterations

(c) PDP, 100 nodes, 1k iterations

(d) PDPTW, 300 calls, 1k iterations

Figure 6.4: Average performance of DRLH, ALNS and URS on each of the problems.



(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

Figure 6.5: Example of the probability of selecting heuristics for DRLH and ALNS.

Even though we have not specifically programmed DRLH to utilize this strategy, it becomes clear by examining Table 6.4 that DRLH has picked up on this strategy when learning to optimize micro-level heuristic selection. Table 6.4 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. Fig. 6.5c 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.

Table 6.4: The percentage of wasted actions of the total number of deterministic heuristics selected averaged over the test set.

(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

Chapter 7

Conclusion and Future Work

In this thesis, we proposed DRLH, a general framework for solving combinatorial optimization problems. In DRLH, we utilize a trained Deep Reinforcement Learning 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 ALNS and URS. Our results show that DRLH is able to select heuristics in a way that achieves better results for almost all of the problem variations compared to ALNS and URS, and DRLH also finds better solutions more quickly than ALNS for most of the problem variations. 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.

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 thesis.

List of Acronyms and Abbreviations

ALNS	Adative Large Neighborhood Search.
ANN	Artificial Neural Network.
CVRP	Capacitated Vehicle Routing Problem.
LNS	Large Neighborhood Search.
MLP	MultiLayer perceptron.
PDP	Pickup and Delivery Problem.
PDPTW	Pickup and Delivery Problem with Time Windows.
PJSP	Parallel Job Scheduling Problem.
PPO	Proximal Policy Optimization.
RL	Reinforcement Learning.
URS	Uniform Random Sampling.

Bibliography

N. Abe, N. K. Verma, C. Apté, and R. Schroko. Cross channel optimized marketing by reinforcement learning. In *KDD*, pages 767–772, 2004. URL `https://doi.org/10.1145/1014052.1016912`.

A. Agarwal, S. Bird, M. Cozowicz, L. Lan, J. Langford, S. Lee, J. Li, D. Melamed, G. Oshri, O. Ribas, S. Sen, and A. Slivkins. A multiworld testing decision service. 06 2016.

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

S. Asta and E. Özcan. An apprenticeship learning hyper-heuristic for vehicle routing in hyflex. Orlando, Florida, 2014. doi: 10.1109/EALS.2014.7009505. URL `http://eprints.nottingham.ac.uk/id/eprint/34399`.

L. Baird and P. Wang. 3d object perception using gradient descent. *Journal of Mathematical Imaging and Vision*, 5(2):111–117, June 1995. doi: 10.1007/bf01250523. URL `https://doi.org/10.1007/bf01250523`.

R. Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503 – 515, 1954. doi: bams/1183519147. URL `https://doi.org/`.

Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35:1798–1828, 08 2013. doi: 10.1109/TPAMI.2013.50.

F. Bertoluzzo and M. Corazza. Reinforcement learning for automated financial trading: Basics and applications. page 197– 213. Springer International Publishing, 2014. doi: 10.1007/978-3-319-04129-2. URL `https://doi.org/10.1007/978-3-319-04129-2`.

E. Burke, M. Hyde, G. Kendall, G. Ochoa, and E. Özcan. *A classification of hyper-heuristic approaches*, pages 449–468. 01 2010a. doi: 10.1007/978-3-319-91086-4_14.

E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward. *A Classification of Hyper-heuristic Approaches*, pages 449–468. Springer US, Boston, MA, 2010b. ISBN 978-1-4419-1665-5. doi: 10.1007/978-1-4419-1665-5_15. URL `https://doi.org/10.1007/978-1-4419-1665-5_15`.

H. Cai, K. Ren, W. Zhang, K. Malialis, J. Wang, Y. Yu, and D. Guo. Real-time bidding by reinforcement learning in display advertising. *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, Feb 2017. doi: 10.1145/3018661.3018702. URL `http://dx.doi.org/10.1145/3018661.3018702`.

X. Chen and Y. Tian. Learning to perform local rewriting for combinatorial optimization. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/131f383b434fdf48079bff1e44e2d9a5-Paper.pdf`.

L. Costa, C. Contardo, and G. Desaulniers. Exact branch-price-and-cut algorithms for vehicle routing. *Transportation Science*, 53(4):946–985, 2019. doi: 10.1287/trsc.2018.0878. URL `https://doi.org/10.1287/trsc.2018.0878`.

P. Cowling, G. Kendall, and E. Soubeiga. A hyperheuristic approach to scheduling a sales summit. In E. Burke and W. Erben, editors, *Practice and Theory of Automated Timetabling III*, pages 176–190, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44629-3.

Y. Crama and M. Schyns. Simulated annealing for complex portfolio selection problems. *European Journal of Operational Research*, 150(3):546–571, 2003. ISSN 0377-2217. doi: https://doi.org/10.1016/S0377-2217(02)00784-1. URL `https://www.sciencedirect.com/science/article/pii/S0377221702007841`. Financial Modelling.

B. Dhingra, L. Li, X. Li, J. Gao, Y.-N. Chen, F. Ahmed, and L. Deng. Towards end-to-end reinforcement learning of dialogue agents for information access, 2017.

T. Dokeroglu, E. Sevinc, T. Kucukyilmaz, and A. Cosar. A survey on new generation metaheuristic algorithms. *Computers & Industrial Engineering*, 137:106040,

2019. ISSN 0360-8352. doi: https://doi.org/10.1016/j.cie.2019.106040. URL `https://www.sciencedirect.com/science/article/pii/S0360835219304991`.

H. Fisher and G. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. *Prentice-Hall, Englewood Cliffs, 225-251.*, 1963.

M. Frank, L. Seeberger, and R. O'Reilly. By carrot or by stick: Cognitive reinforcement learning in parkinsonism. *Science (New York, N.Y.)*, 306:1940–3, 01 2005. doi: 10.1126/science.1102941.

I. J. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* MIT Press, Cambridge, MA, USA, 2016. `http://www.deeplearningbook.org`.

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

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

M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.

K. L. Hoffman and T. K. Ralphs. *Integer and Combinatorial Optimization*, pages 771–783. Springer US, Boston, MA, 2013. ISBN 978-1-4419-1153-7. doi: 10.1007/978-1-4419-1153-7_129. URL `https://doi.org/10.1007/978-1-4419-1153-7_129`.

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

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

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. ISSN 0036-8075. doi: 10.1126/science.220.4598.671. URL https://science.sciencemag.org/content/220/4598/671.

J. Kober, J. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32:1238–1274, 09 2013. doi: 10.1177/0278364913495721.

V. Konda and J. Tsitsiklis. Actor-critic algorithms. *Society for Industrial and Applied Mathematics*, 42, 04 2001.

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

S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies, 2016.

S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5):421–436, 2018. doi: 10.1177/0278364917710318. URL https://doi.org/10.1177/0278364917710318.

H. Lu, X. Zhang, and S. Yang. A learning-based iterative method for solving vehicle routing problems. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=BJe1334YDH.

J. Lundgren, M. Rönnqvist, and P. Värbrand. *Optimeringslära*. Studentlitteratur, Lund, 2. uppl. edition, 2003. ISBN 9144031041.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015. doi: 10.1038/nature14236.

M. Nazari, A. Oroojlooy, L. Snyder, and M. Takac. Reinforcement learning for solving the vehicle routing problem. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/9fb4651c05b2ed70fba5afe0b039a550-Paper.pdf.

E. Özcan, M. Misir, G. Ochoa, and E. Burke. A reinforcement learning - great-deluge hyper-heuristic for examination timetabling. *Int. J. Appl. Metaheuristic Comput.*, 1:39–59, 2010.

R. Paulus, C. Xiong, and R. Socher. A deep reinforced model for abstractive summarization, 2017.

D. Pisinger and S. Ropke. A general heuristic for vehicle routing problems. *Computers Operations Research*, 34(8):2403–2435, 2007. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2005.09.012. URL `https://www.sciencedirect.com/science/article/pii/S0305054805003023`.

M. Popova, O. Isayev, and A. Tropsha. Deep reinforcement learning for de novo drug design. *Science Advances*, 4(7):eaap7885, Jul 2018. ISSN 2375-2548. doi: 10.1126/sciadv.aap7885. URL `http://dx.doi.org/10.1126/sciadv.aap7885`.

S. Ropke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, Nov. 2006. ISSN 1526-5447. doi: 10.1287/trsc.1050.0135. URL `https://doi.org/10.1287/trsc.1050.0135`.

A. Sallab, M. Abdou, E. Perot, and S. Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017(19):70–76, Jan 2017. ISSN 2470-1173. doi: 10.2352/issn.2470-1173.2017.19.avm-023. URL `http://dx.doi.org/10.2352/ISSN.2470-1173.2017.19.AVM-023`.

A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume B. 01 2003.

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

J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In M. Maher and J.-F. Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 417–431, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-49481-2.

D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016. doi: 10.1038/nature16961.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.

K. Sörensen and F. Glover. *Metaheuristics*, pages 960–970. 01 2013. ISBN 978-1-4419-1137-7. doi: 10.1007/978-1-4419-1153-7_1167.

R. Turkeš, K. Sörensen, and L. M. Hvattum. Meta-analysis of metaheuristics: Quantifying the effect of adaptiveness in adaptive large neighborhood search. *European Journal of Operational Research*, 292(2):423–442, 2021. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2020.10.045. URL `https://www.sciencedirect.com/science/article/pii/S037722172030936X`.

R. Tyasnurita, E. Özcan, A. Shahriar, and R. John. Improving performance of a hyperheuristic using a multilayer perceptron for vehicle routing. Exeter, UK, 2015. URL `http://eprints.nottingham.ac.uk/id/eprint/45707`.

H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning, 2015.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2017.

Y. Zhao, M. Kosorok, and D. Zeng. Reinforcement learning design for cancer clinical trials. *Statistics in medicine*, 28 26:3294–315, 2009.

Appendix A

Experiments on Different Reward Functions

A.1 R_t^{PM}

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

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.

Fig. A.1 illustrates the distribution of minimum costs found on the on the PDP100 test set after 1000 and 10000 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 value for both iteration variations, with higher cost gathered around the median value compared to the model trained with R_t^{PM}. This makes the R_t^{5310} reward function more reliable to perform close to median, and we therefore decided to use the R_t^{5310} reward function in this thesis.

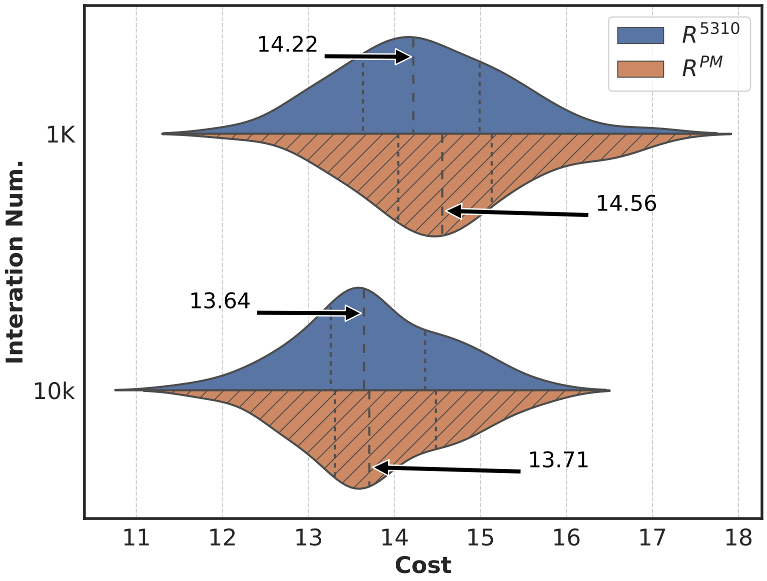


Figure A.1: Comparison of the two reward functions.

A.2 R_t^{MC}

$$R_t^{MC} = \begin{cases} \frac{f(l_{best}) - f(l')}{f(l_{best})} \end{cases} \tag{A.2}$$

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 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.

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 10000 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

Appendix B

Extended Set of Heuristics

Tables B.1, B.2 and B.3 list the extended set of heuristics built up from 14 removal operators, 10 insertion operators and 2 additional operators, 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 B.1: 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	Removes between 30-40 elements that has been removed the least
Remove_one_vehicle	Removes all the elements in one vehicle
Remove_two_vehicles	Removes all the elements in one vehicle

Table B.2: 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.

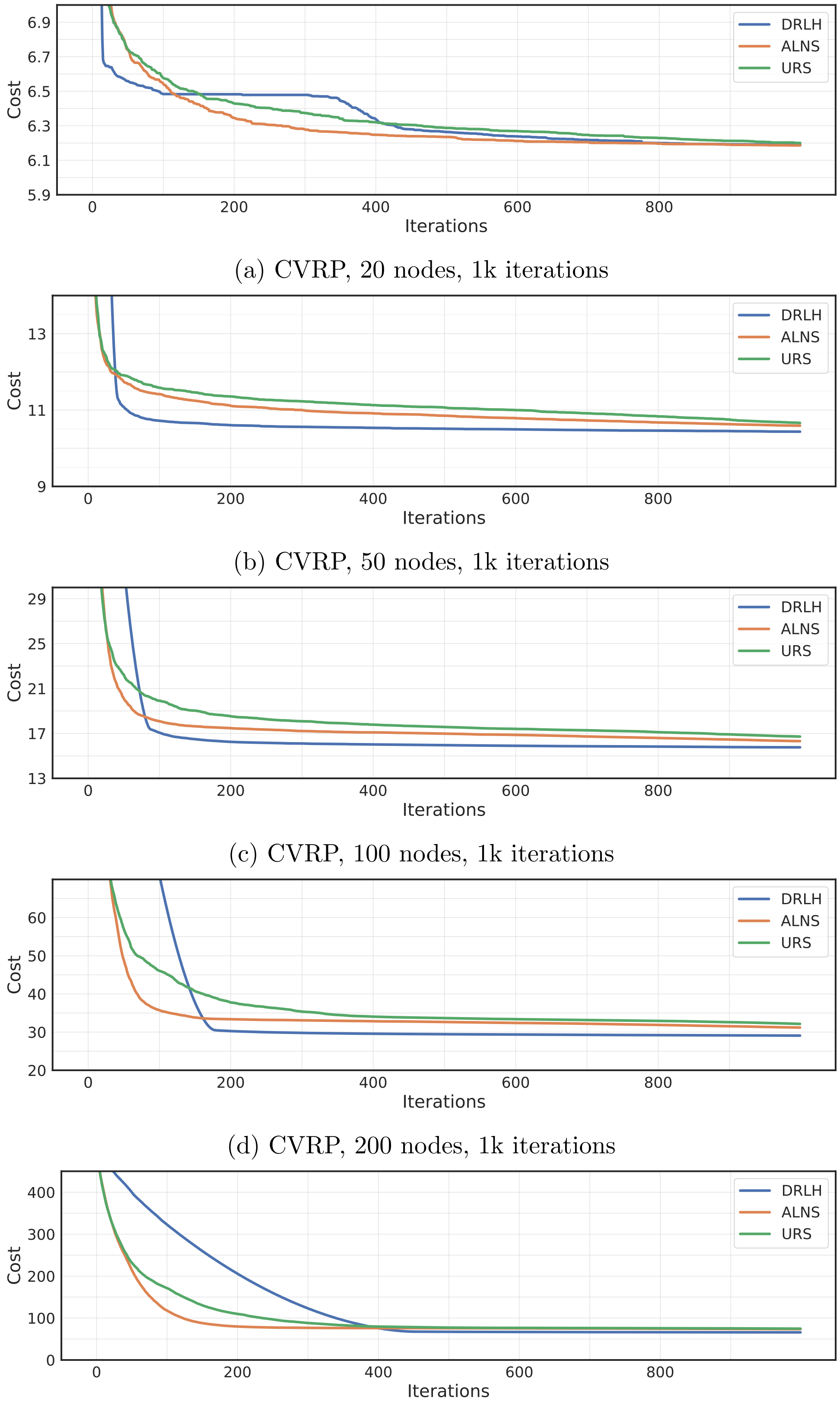
Table B.3: List of extended additional heuristics

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 l for the element that achieves the minimum cost $f(l')$.
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 C.1, C.2, C.3 and C.4 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.



(a) CVRP, 20 nodes, 1k iterations

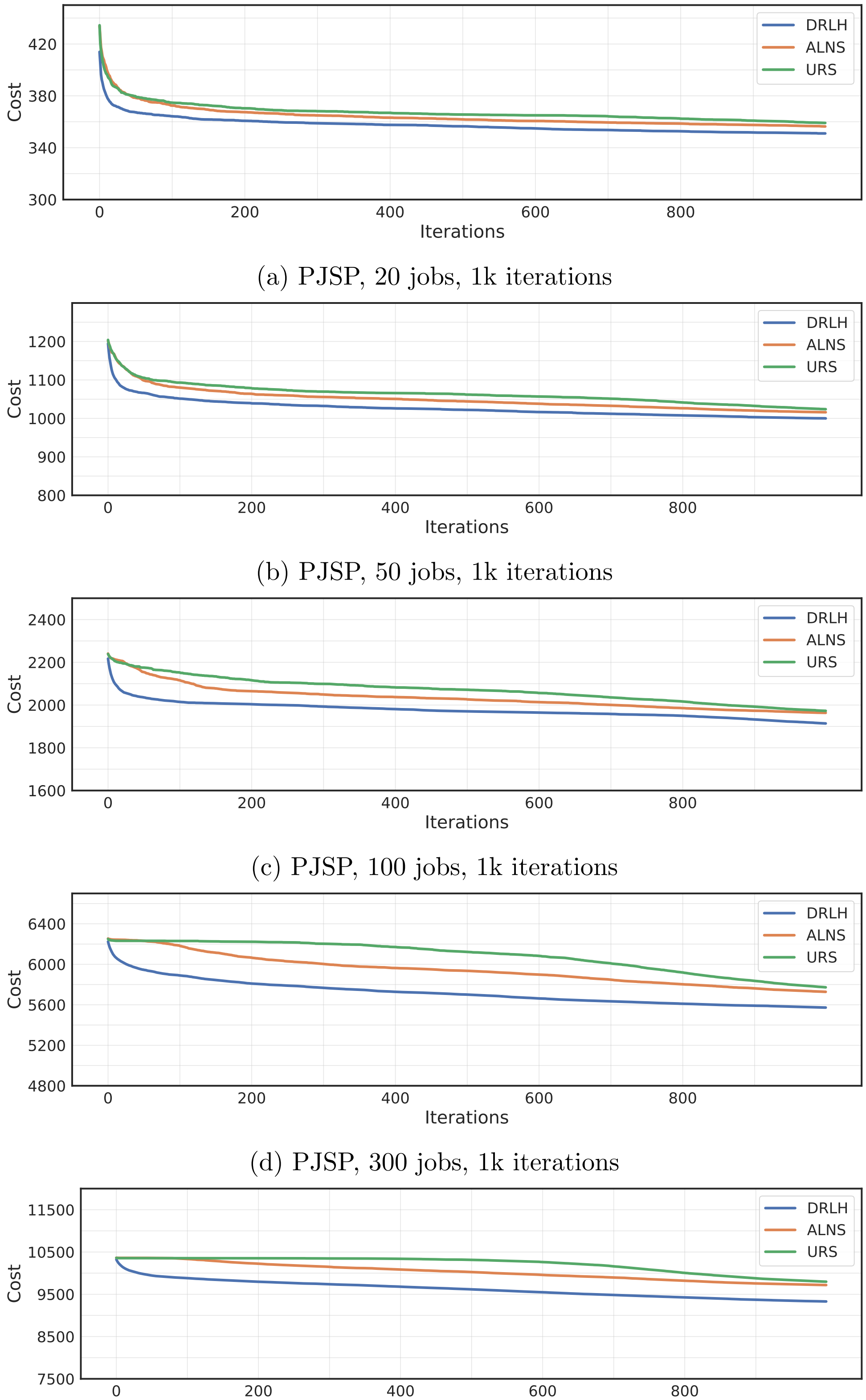
(b) CVRP, 50 nodes, 1k iterations

(c) CVRP, 100 nodes, 1k iterations

(d) CVRP, 200 nodes, 1k iterations

(e) CVRP, 500 nodes, 1k iterations

Figure C.1: Average performance of DRLH, ALNS and URS on CVRP.



(a) PJSP, 20 jobs, 1k iterations

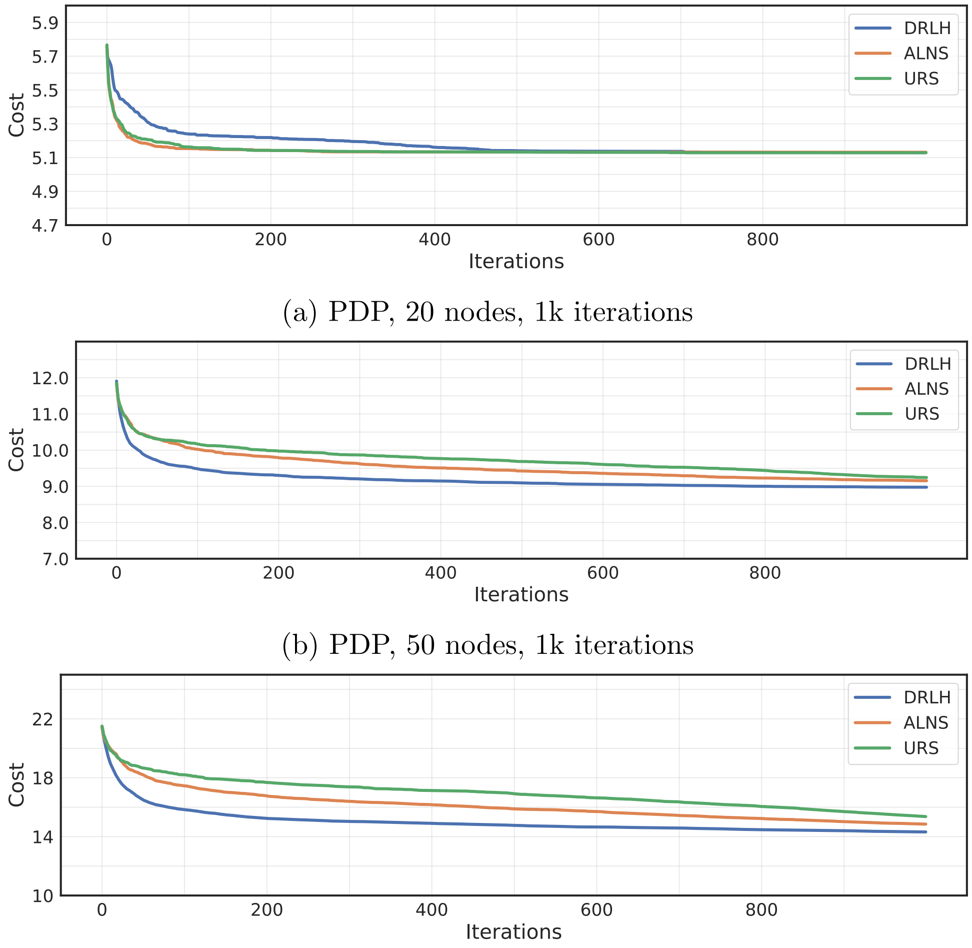
(b) PJSP, 50 jobs, 1k iterations

(c) PJSP, 100 jobs, 1k iterations

(d) PJSP, 300 jobs, 1k iterations

(e) PJSP, 500 jobs, 1k iterations

Figure C.2: Average performance of DRLH, ALNS and URS on PJSP.

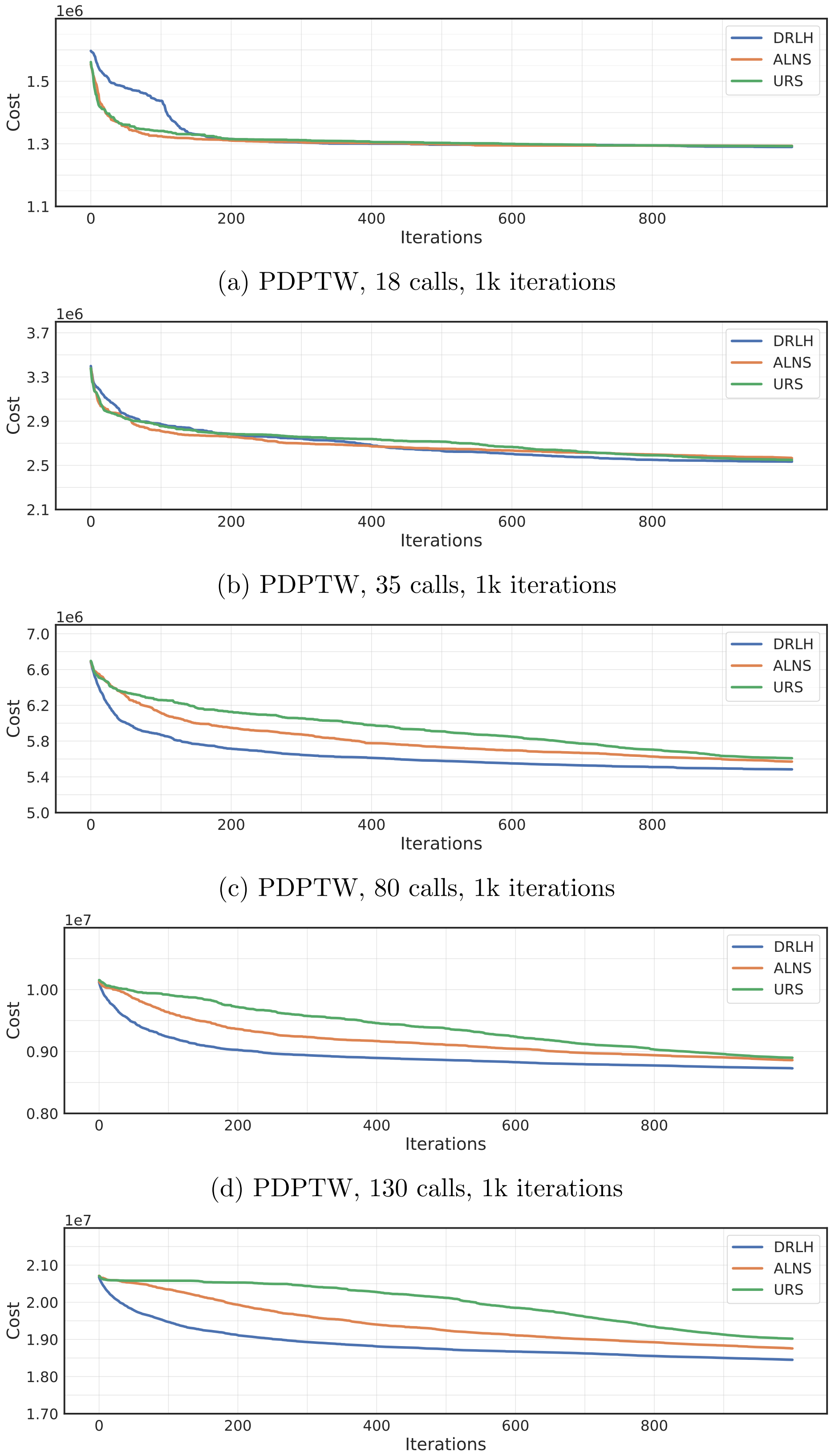


(a) PDP, 20 nodes, 1k iterations

(b) PDP, 50 nodes, 1k iterations

(c) PDP, 100 nodes, 1k iterations

Figure C.3: Average performance of DRLH, ALNS and URS on PDP.



(a) PDPTW, 18 calls, 1k iterations

(b) PDPTW, 35 calls, 1k iterations

(c) PDPTW, 80 calls, 1k iterations

(d) PDPTW, 130 calls, 1k iterations

(e) PDPTW, 300 calls, 1k iterations

Figure C.4: Average performance of DRLH, ALNS and URS on PDPTW.