University of Bergen
Department of Informatics

DRLMA: An Intelligent Move Acceptance for Combinatorial Optimization Problems based on Deep Reinforcement Learning

Author: Eskil Hamre Isaksen
Supervisor: Ahmad Hemmati
Co-supervisor: Ramin Hasibi

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2023

Abstract

Numerous heuristic solution methods have been developed to tackle combinatorial optimization problems, often customized for specific problem domains and use-cases where they exhibit remarkable performance. However, their effectiveness diminishes significantly when applied to problem domains for which they were not originally designed, showcasing poor generalization capabilities. In contrast, metaheuristics are higher-level heuristics solution methods that aim to be applicable to a wide range of different problems. Perturbative metaheuristics operate by traversing the solution space through iterative application of modifications induced by low-level heuristics. This process continues until a specified stopping criteria is met, enabling the method to efficiently explore and refine solutions. A central aspect of these search-based methods is the move acceptance scheme, which determines whether or not the suggested modification is to be applied. The Simulated Annealing acceptance criteria, for instance, occasionally accepts uphill moves, or worse solutions, in order to explore the space of solutions and help the search escape local optima. In this thesis we propose Deep Reinforcement Learning Move Acceptance (DRLMA), a general move acceptance framework that leverages Deep Reinforcement Learning into the acceptance decision. A Deep RL agent is trained using problem-independent search information, enabling it to learn high-level acceptance strategies regardless of the specific combinatorial optimization problem at hand. We show that by replacing the Simulated Annealing acceptance criteria with DRLMA in two different heuristic selection frameworks, namely Adaptive Large Neighborhood Search (ALNS) and Deep Reinforcement Learning Hyperheuristic (DRLH), we are generally able to improve the performance of the respective search methods, the degree of improvement ranging from only slightly in the worst cases to considerably in the best cases.

Acknowledgements

I would like to express my heartfelt appreciation to my supervisors, Ahmad Hemmati and Ramin Hasibi, for their invaluable guidance and support throughout my master's studies at the University of Bergen. I am deeply grateful for the insightful discussions we have had, for their encouragement to explore my own ideas and for their trust in my ability to manage my time effectively. I am indebted to them both for the time and effort they have invested in my development, and for being an understanding and resourceful presence throughout my journey.

Furthermore, I am immensely grateful to all those who have have supported me throughout the completion of this master thesis. While the research itself has been a fun and fulfilling endeavor, it is the camaraderie and steadfast support of my peers that have truly enriched this experience. I would especially like to express my gratitude to Thorarinn Gunnarsonn in this regard, for without his presence my time at UiB would have been much less enjoyable. I'd also like to give a special shout-out to Audun Ljone Henriksen and Herman Jangsett Mostein for our enlightening exchanges. Without them, I'm pretty certain my upcoming summer vacation would've been seriously cut short. Thanks for keeping it intellectually stimulating and saving my precious downtime! I would also like to thank my family and friends who have contributed immensely to keeping me motivated throughout this entire journey and for providing me with healthy and much-needed distractions along the way. Lastly, a big thank you to my incredible girlfriend Pauline, for your unwavering understanding and support. I am forever thankful for the patience you have displayed by sticking with me these last few months.

Eskil Hamre Isaksen
Tuesday 27$^{\text{th}}$ June, 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

Combinatorial optimization problems encompass a wide range of problems that involves finding an arrangement or a selection of elements from a finite set such that an objective function is optimized. Typical combinatorial optimization problems include the Traveling Salesman Problem (TSP), the Minimum Spanning Tree Problem (MST) and the Knapsack Problem (KP). Many of these problems cannot be solved in polynomial time, and exhaustive search-based methods becomes intractable very quickly as the size of the problem grows. For this reason one often resort to approximation solution methods instead, called heuristics, which deal with quickly finding solutions of sub-optimal but satisfactory quality.

Metaheuristics, a powerful class of solution methods, consists of algorithmic frameworks that each provide a set of rules or strategies that make up a heuristic optimization algorithm. Examples of classical metaheuristics are Genetic Algorithm (GA), Tabu Search (TS), Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO) and Simulated Annealing (SA). In recent times, Adaptive Large Neighborhood Search (ALNS) (Ropke and Pisinger, 2006) is a frequently used metaheuristic that has proven to be quite effective on several combinatorial optimization problems. In ALNS, a set of combined destroy-repair operators with different attributes is used to repeatedly modify a solution, exploring parts of the solution space and discovering better solutions along the way. When the search ends, the solution with the best objective value is yielded. The main contribution of ALNS lies in its adaptive layer which determines the operator selection

process. During the search, each operator has scores associated with them that reflects the effectiveness of the operator in the most recent part of the search. These scores makes up the operator selection probabilities, such that recently well-performing operators are more likely to be selected in the future.

ALNS also has some shortcomings, however. Firstly, Turkeš et al. (2021) claim that the effectiveness of the adaptive layer of ALNS has been widely overstated, further claiming that it only has a small impact on the objective value in general. Secondly, the ALNS search is divided into segments consisting of a certain number of search iterations, in which the operator scores - and thus the selection probabilities - stays constant. The scores are only updated in between two segments. Although the decision making capabilities possess a certain level adaptability on a "macro-level", it remains quite rigid on a "micro-level". From iteration to iteration within a segment, the operator choice is randomly sampled according to the fixed operator probabilities. This design employs a considerable limitation on the decision making capabilities of ALNS.

Kallestad et al. (2023) proposed Deep Reinforcement Learning Hyperheuristic (DRLH) to mitigate the shortcomings of ALNS. In their contribution they replace the adaptive layer of ALNS with a deep reinforcement learning (RL) agent. The RL agent is responsible for selecting which operator to apply to the solution at every iteration of the search. The agent is trained to use available problem-independent information regarding the search, such as the difference in objective value between past solutions and the number of iterations since the solution was last changed, to select an operator so as to maximize a long-term reward function. This makes DRLH adaptable on a "micro-level", at each iteration of the search being able to recognize problem-independent situations and efficiently map them to a certain operator. As opposed to ALNS, the operator selection probabilities may change quickly from iteration to iteration since the agent is able to immediately adapt to new situations. The authors show that DRLH is able to significantly outperform ALNS on four different combinatorial optimization problems, clearly displaying the generalization capabilities of their contribution.

An entirely different aspect of these iterative solution methods is the *move acceptance*, also known as the *acceptance criteria*, which is combined with the operator selection scheme. The selected operator modifies the incumbent solution into a new solution, called the candidate solution. This "move" in the solution space is merely a suggestion. The move acceptance decides whether the move should be accepted or rejected, and so if the candidate is accepted, it takes the place as the incumbent solution. It is generally considered wise to accept downhill moves (improvements), so the defining trait of

different move acceptances is usually their strategy when dealing with uphill moves (non-improvements). The move acceptance turns out to be a very important decision, vital to a successful search algorithm. Classical move acceptances include Hill climbing, Record-To-Record Travel, Great Deluge and the Simulated Annealing acceptance criteria, which all appear quite frequently in the literature.

In a their cross-domain performance comparison of different move acceptances, Jackson et al. (2018) show the Simulated Annealing acceptance criteria to be the most effective one in general terms. In Simulated Annealing, the acceptance decision is based on a stochastic framework called the Metropolis criteria which depends on the change in objective value and the temperature which decreases in time (Kirkpatrick et al., 1983). The overall strategy of the Simulated Annealing acceptance criteria is to gradually decrease the probability of accepting uphill moves as time progresses. By frequently allowing uphill moves at the beginning of the search, it is able to explore diverse regions of the solution space, and as the search progresses the strategy is progressively shifted to favour downhill moves only - to converge towards as good solutions as possible. Both ALNS and DRLH use the Simulated Annealing acceptance criteria.

Although generally considered to be a strong move acceptance, we argue that it isn't flawless. Its performance can be sensitive to the selection of certain hyperparameters, such as the initial temperature and the cooling schedule. Different problems may require different hyperparameter configurations, and careful experimentation and tuning is often required to achieve good performance. Moreover, most move acceptances in literature, including the Simulated Annealing acceptance criteria, has the property that acceptance is based on only a narrow fraction of the information available regarding the search. Inspired by the works presented by Kallestad et al. (2023), we believe that more sophisticated move acceptance can be developed by allowing more search information to be taken into consideration.

In this thesis we propose **Deep Reinforcement Learning Move Acceptance (DRLMA)**, a general move acceptance for solving combinatorial optimization problems. The move acceptance consists of a Deep RL agent which is trained using a popular Deep RL technique called Proximal Policy Optimization (Schulman et al., 2017). DRLMA is trained using problem-independent information about the search process as its state representation, combined with a problem-independent reward function that encourages the search to locate better solutions in the near future. By removing DRLMA from any problem-specific details we allow the framework to be applied to different combinatorial optimization problems, which we show experimentally by solving two different combinatorial optimization problems, namely the Capacitated Vehicle Routing Problem (CVRP)

and the Parallel Job Scheduling Problem (PJSP). Training DRLMA makes it adaptable to different problem settings by allowing it to develop a problem-specific move acceptance strategy for each individual problem setting. Both the state representation, the reward function and remaining hyperparameters stays constant across the two problems, removing the need for problem-specific parameter tuning which is usually required to achieve good performance when using ALNS and Simulated Annealing. Furthermore, we combine DRLMA with two different hyperheuristic frameworks, showing that DRLMA displays a general tendency to outperform the Simulated Annealing acceptance criteria in different operator-selection settings. Most importantly perhaps, this also shows that the performance of DRLH can be further improved upon by replacing the Simulated Annealing acceptance criteria with DRLMA.

1.2 Thesis outline

Chapter 2 - Background and Related Work touches upon the theoretical aspects of combinatorial optimization and reinforcement learning that we find relevant to this thesis. This section also covers previous work related to solving combinatorial optimization problems by embedding elements of reinforcement learning into the optimization algorithm.

Chapter 3 - Problem Sets briefly describes the two combinatorial optimization problems that we use to display the effectiveness of DRLMA.

Chapter 4 - DRLMA introduces the Deep Reinforcement Learning Move Acceptance, the main contribution of this thesis. We describe the framework in detail, along with the operators used by all the solution methods.

Chapter 5 - Experimental Setup describes the experimental details of this thesis, including the hardware and software used to set up and run the experiments, information regarding the baseline methods used to compare the performance of DRLMA, experimental hyperparameters and reward function specifics, and the generation of the problem instance datasets.

Chapter 6 - Results displays the experimental statistics and findings, along with a discussion of their significance.

Chapter 8 - Conclusion and Future Work summarizes the work and main findings of this thesis, followed by a brief proposition of future work related to the thesis based what we experienced when developing it.

Chapter 2

Background and Related Work

2.1 Combinatorial Optimization

Optimization is the process of making a design or decision as effective as possible relative to a set of criteria or constraints (Kelley, 2010). Combinatorial optimization is a subfield within optimization concerned with finding an optimal object from a finite set of objects (Schrijver, 2003). These objects, or solutions, are *discrete* in their nature, consisting either of an arrangement or a selection of elements originating from a finite set of elements. These kinds of problems arise in many different disciplines, such as operations research, artificial intelligence, algorithmic theory, bioinformatics, transportation and electronic commerce. Some prominent examples of combinatorial optimization problems include Vehicle Routing Problems, Graph Colouring Problems, Knapsack Problems, Scheduling Problems and Bin Packing Problems. A combinatorial optimization problem is based on an objective function and a set of logical conditions or constraints. A solution that satisfies these logical conditions are deemed *feasible* or *valid*, and feasible solutions are deemed comparable in terms of their objective function value. An *optimal solution* is the solution among the finite set of feasible solutions with either the highest or lowest objective, depending upon whether the problem is a maximization or minimization problem, respectively (Hoos and Stützle, 2005). A solution to the Scheduling Problem, for instance, assigns each worker or machine an arrangement of jobs, such that all the jobs are handled exactly once.

It turns out that being able efficiently solve combinatorial optimization problems is of great economic importance. A food delivery company, for instance, must be able schedule

its available drivers in a manner such that delivery time is minimized and/or profit is maximized, and an airline company would want to set up schedules for their crew such that the total operational cost is minimized. The difference between improvisational on-the-fly planning directed by humans and advanced algorithmic planning to find the optimal course of actions can mean the difference between great economic success and bankruptcy for a company (Hoffman and Ralphs, 2013).

There exists a fundamental divide in combinatorial optimization problems based on whether they can be solved in polynomial time (meaning efficiently) or not. Problems that are concerned with finding shortest paths, optimal flows and spanning trees are examples of problems where the optimal solution may be obtained efficiently. On the other hand, many of the combinatorial optimization problems that frequently appear in the real world, such as the Traveling Salesman Problems, Vehicle Routing Problems and Scheduling Problems, belong to the complexity class known as NP-Complete. Simply put this means that no efficient optimization algorithm can be constructed (assuming P $\neq$ NP, a reasonable assumption which we won't linger on in this thesis). Furthermore, exhaustive search-based solution methods for these kinds of problems are generally considered intractable: real-world instances of these problems are usually so big that a complete enumeration of the solution space, the set of all feasible solutions, simply would take too long a time. Throughout the rest of this thesis, the term *combinatorial optimization problems* refers to the latter type of problems. There exists several solution methods for these kinds of combinatorial optimization problems, and we will touch upon some of them in the following section.

2.2 Solution Methods

In this thesis we find it appropriate to divide the solution methods of combinatorial optimization problems into two categories, namely *exact approaches* and *heuristic approaches*. As our thesis' main contribution is a heuristic approach, we will explore this category most in-depth. However, first we shortly introduce the concept of exact approaches to emphasize the existence of alternatives to heuristic approaches, and most importantly to illustrate the need for heuristic approaches.

2.2.1 Exact approach

When solving a combinatorial optimization problem, exact approaches offers a guarantee of obtaining the optimal solution. The downside is that these approaches usually have to

exhaustively explore large parts of the solution space, severely impacting the algorithmic run time when dealing with large-scale and tightly constrained problems. One commonly used exact approach is the *branch-and-bound* method, which uses a tree search to divide the solution space into smaller subspaces and strategically excludes those regions which cannot lead to a better solution (Morrison et al., 2016). Although a complete solution space enumeration is avoided, this method still becomes time-consuming for problems of realistic size, limiting its practicality. When used in practise, the search is typically stopped when a limit is reached, such as a time limit or maximum number of solutions visited - turning this solution method into a heuristic. Other examples of exact approaches are branch-and-bound, branch-and-price and dynamic programming.

2.2.2 Heuristic approach

As opposed to exact algorithms, heuristic approaches do not guarantee to find an optimal solution. The main strength of heuristic approaches is that they find "good enough" solutions with much less computational effort than exact approaches. For large-scale problems where exact approaches becomes intractable, heuristic approaches is the only alternative. For instance, Drezner et al. (2005) show that exact methods are unable to solve instances of the Quadratic Assignment Problem to optimally when the problem size grows beyond 30-40, whereas heuristic approaches manages to quickly find high quality solutions. A lot of research has been conducted by the optimization community on developing heuristic techniques that produce as good solutions as possible. Before diving further into the different techniques, we find it useful to divide heuristic approaches into two categories: *constructive heuristics*, which builds a solution from scratch by iteratively adding elements, and *perturbative heuristics*, which starts out which some initial solution and repeatedly applies modifications in order to improve it. Of these, we will explain the latter category in greater detail, since it encompasses this thesis' contribution.

Constructive Heuristics

Constructive heuristics build a solution by repeatedly adding elements until a complete solution is obtained. These heuristic approaches are generally very fast and produce solutions with higher quality than random construction approaches, but quality-wise they usually fall way short compared to exact approaches and even perturbative heuristics since they do not perform local modifications to the constructed solution. In fact, due to

the speed of constructive heuristics they are often employed to create initial solutions for perturbative heuristics approaches, which then further improves upon the initial solution. An example of a simple constructive heuristic is the greedy one: in TSP, for instance, a solution can be constructed by starting with an arbitrary city and iteratively select the closest unvisited city to add to the solution until all cities has been visited.

Perturbative Heuristics

Perturbative heuristics, also called local search methods, iteratively apply a rule-based and potentially stochastic modification to the solution l, producing a neighboring solution l'. The functional core performing the modification itself has several names in the literature, ranging from *heuristics*, *move operators* or simply just *operators*. In this thesis we use both the terms "heuristic" and "operator" to refer to this functional core.

Recall that combinatorial optimization can be viewed as searching for the best object (solution) in a finite set of objects (the solution space). Exact solution methods, such as branch-and-bound, does this by visiting a large portion of the solutions in the solution space. Perturbative heuristics, however, seeks to drastically limit the number of visited solutions to speed up the search considerably. The goal of any perturbative heuristic is thus to "make every second count" - to obtain an as high-quality solution as possible under the constraint of being limited to a tiny fraction of the solution space.

The design of a heuristic (as in operator) reflects a certain search-strategic idea. For instance, a greedy heuristic produces solutions closer to local optimum in the solution space. Solely relying on such a heuristic will trap the search in the closest local optimum, generally leading to solutions of poor quality. When designing a heuristic, a common idea is to include a degree of randomness within it. The *neighborhood* of a solution is the set of all solutions that can be obtained by applying the heuristic a single time. Stochastic heuristics has a larger neighborhoods than deterministic heuristics, and is much less likely to get stuck in local optima. A different design aspect is the question of how much the heuristic should modify the solution. Small-scale heuristics perform small modifications to the solution and are generally computationally inexpensive, allowing the search method to explore a vast set of solutions in a short amount of time. However, search methods that rely on repeated small modifications have difficulties moving from one promising part of the solution space to the next.

A core aspect of perturbative heuristic methods is the stopping criteria. This typically puts a limit on either the amount of perturbations or on the running time of the search.

When the stopping criteria is reached, the algorithm terminates and yields the global best solution found. The stopping criteria we adapt in this thesis limits the amount of perturbations, or search iterations.

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). As with heuristics, metaheuristics solution methods can be divided into *constructive* and *perturbative* metaheuristics. In the remainder of this thesis we use the term "metaheuristics" to refer to "perturbative metaheuristics", as they appear most frequent in the literature and is highly relevant to this thesis.

Metaheuristics are regarded as problem-independent, as they make few assumptions on the underlying details of the problem being solved. This makes metaheuristics very flexible optimization framework suitable for solving a wide range of different combinatorial optimization problems. A central aspect of metaheuristics is that they search within the solution space, comparing the objective of different solutions encountered to guide the search onward. The search process can be summarized as iteratively modifying solutions by applying various operators in some rule-based fashion, allowing the search to explore different regions of the solution space, escape local optima and move towards promising solutions. Examples of metaheuristics that frequently appear in the literature are Genetic Algorithms, Tabu Search, Simulated Annealing, Variable Neighborhood Search and Adaptive Large Neighborhood Search.

Metaheuristics can further be classified in several dimensions, such as trajectory methods versus discontinuous methods, single-point search versus population-based search, one vs. various neighborhood structures (Birattari et al., 2003). In this thesis we focus on metaheuristics that embeds *move acceptance methods* into a single-point, trajectory-based search framework. In literature, these kinds of metaheuristics are also known as *hyperheuristics* (Burke et al., 2013), a class of search methods we will describe in further detail in Section 2.3. The move acceptance determines if the search should move to the suggested solution or stay at the incumbent one, thus playing a vital role guiding the search in the correct direction. This will be explained in greater detail in Section 2.4.1.

Intensification and diversification

Metaheuristics are constrained to visit only a miniature fraction of the solution space by moving from one solution to next in a trajectory-like manner until the stopping criteria is met. The central question in these kinds of metaheuristics is how to guide the search-trajectory to obtain the best possible solution. A challenging aspect of this question is finding a good balance between two conflicting notions throughout the search, namely *intensification* and *diversification*. Intuitively, parts of the solution space deemed promising should be explored as much as possible, increasing the chance of discovering good solutions. This notion is known as intensification. At the same time, the search should also cover as large portion of the solution space as possible to prevent the search from being confined to a certain region of the solution space. This notion is known as diversification. Obviously, both notions have their place in the search, and the metaheuristic should balance the two in a sensible manner.

2.3 Hyperheuristics

As mentioned, the design of an operator, or heuristic, should reflect a certain strategic idea. It turns out that the quality of the search method might improve by gathering multiple lower-level heuristics with different attributes and use-cases into a *heuristic pool*, and then strategically alternate between which heuristic to employ. Hyperheuristics are heuristic search methods that aim to intelligently direct the selection process (or generation process) of lower-level heuristics throughout the search, based on problem-independent information available. In simpler (and slightly less accurate) terms, hyperheuristics are *heuristics to choose heuristics* (Cowling et al., 2001).

Unlike metaheuristics which search within the solution space, hyperheuristics search within the search space of heuristics. In this sense, hyperheuristics aim to operate at a level of abstraction above the problem instance and even the class of problems being solved. In other words, a *domain barrier* is set up between the heuristic selection process and the dynamics of the underlying problem, as displayed in Figure 2.1. This reflects the general goal of the hyperheuristic approach: to discover a generally applicable methodology capable of solving a wide range of problem classes, at the cost of slightly subpar solution qualities compared to problem-specific heuristic methods. As opposed to problem-specific heuristic methods, the low-level heuristics typically used in hyperheuristic frameworks are easy to implement across different problem classes, requiring little-to-none domain knowledge.

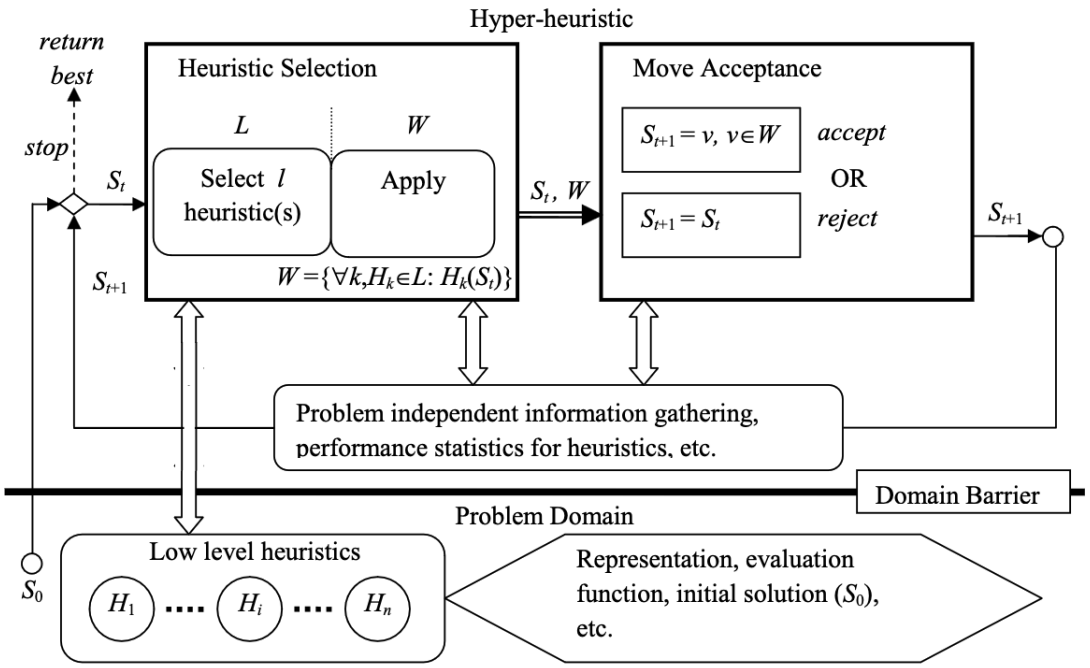


Figure 2.1: The general hyperheuristic framework. At time-step t, the incumbent solution S_t is passed to the Heuristic Selection part of the algorithm, where a set of heuristics L is selected and applied onto S_t, producing a set of candidate solutions W. S_t and W is further passed onto the Move Acceptance, which either accepts a solution in W or rejects W entirely, resulting in the incumbent solution for time-step $t + 1$. The domain barrier separates the hyperheuristic framework from low-level problem-dependent details. Image taken from Özcan et al. (2010).

2.4 Adaptive Large Neighborhood Search

Large Neighborhood Search (LNS), proposed by Shaw (1997), was among the first solution methods that made large-scale solution modification tractable through the use of a combined destroy and repair heuristic. Unlike typical small-scale local search heuristics, destroy and repair heuristics can rearrange 30-40% of the solution components, producing vastly different solutions. This allows the search to explore a completely different part of the solution space with a single operation. This is useful in complex combinatorial optimization problems which has to meet many constraints, such as routing problems (Schrimpf et al., 2000). Although a destroy and repair heuristic is a much slower operation, in practise it seems to work at least as effective as solution methods using small-scale heuristics when given the same amount of time. Building upon the idea of large neighborhoods, Adaptive Large Neighborhood Search (ALNS) expands upon LNS, making it into a hyperheuristic by providing a selection strategy for multiple destroy and repair heuristics (Ropke and Pisinger, 2006).

The selection process is stochastic, and each heuristic has an associated probability of being selected. At the beginning of the search, all heuristics are assigned equal probabilities. During the search the probabilities are dynamically updated based on the recent performance of each heuristic. Thus, selection of recently well-performing heuristics becomes more likely. The search is divided into many *segments* with a predefined size, in which heuristic probabilities stay stationary. When a heuristic is selected and applied to the incumbent solution l to produce a candidate solution l', it receives an immediate score based on a score function on the form:

$$\psi = max \begin{cases} \omega_1 & \text{if } f(l') < f(l_{best}), \\ \omega_2 & \text{if } f(l') < f(l), \\ \omega_3 & \text{if accept}(l'), \\ \omega_4 & \text{else} \end{cases} \tag{2.1}$$

where ω_1, ω_2, ω_3 and ω_4 are parameters set by the client. Usually $\omega_1 \geq \omega_2 \geq \omega_3 \geq \omega_4 \geq 0$. Thus, the higher the value the more successful the heuristic. During the segment the search keeps track of the total score for each heuristic. When the segment ends, the probability distribution is updated such that high-scoring heuristics become more likely in the new segment. The scores resets, and a new segment starts. This is the *adaptive* part of ALNS.

Two hyperparameters determines the effectiveness of this solution method, namely the *reaction factor* and the *segment size*. The reaction factor determines the degree in which the scores from the last segment should affect the new probability distribution, like a trade-off between emphasis on past performance and most recent performance. The segment size determines the number of optimization steps the algorithm should perform before the probability distribution is updated. Longer segments means less frequent updates to the probability distribution, increasing the time it takes for valuable information to influence the probabilities. However, once an update is performed it is less affected by stochastic noise. In the same manner, a low reaction factor mean a more conservative probability distribution update, meaning it takes longer to achieve optimal probabilities. At the same time it avoids potentially harmful updates caused by stochastic noise in the search process.

2.4.1 Simulated Annealing Acceptance Criteria

The purpose of the move acceptance, also called the acceptance criteria, is to aid the search in finding a sensible balance between intensification and diversification and help the search escape local minima. Observe in Eq.2.1 that heuristics receive a small score if an uphill move is accepted. The idea is that heuristics should also be slightly awarded for producing solutions that pass the acceptance criteria, so as to not solely encourage the selection of intensifying heuristics. But how should the acceptance criteria be designed so that it aids the search in a best possible way?

Ropke and Pisinger (2006), the authors who proposed ALNS, suggest using the same move acceptance presented in the Simulated Annealing (SA) metaheuristic, introduced by Kirkpatrick et al. (1983). This move acceptance, inspired from physics, stochastically accepts uphill moves with a probability based on the *Boltzmann function*. The details surrounding this move acceptance vary slightly in literature, and so we present the version that Kallestad et al. (2023) used in their experimentation and that consequentially we use in our own experiments. The SA acceptance criteria derives the probability of accepting the candidate solution l' based on the following formula:

$$P_{\text{Simulated Annealing}}(\text{accept}) = \begin{cases} 1.0 & f(l') < f(l), \\ 0.0 & \text{if } l' \text{ has been encountered before,} \\ p_{\text{warmup}} & \text{if } \textit{warm-up phase}, \\ e^{-\frac{\Delta E}{T}} & \text{otherwise} \end{cases}$$

where the given order is regarded as the order of precedence - the first satisfied condition determines the probability.

The Boltzmann function $e^{-\frac{\Delta E}{T}}$ contains two parameters, the first being the difference in cost between the incumbent l and new solution l', denoted $\Delta E = f(l') - f(l)$. The cost difference term ensures that small uphill steps is more likely than large ones. The second parameter is called the *temperature* T, which is gradually decreases throughout the search. At the beginning of the search the algorithm is more likely to perform larger uphill moves, allowing exploration of large portions of the solution space. When temperature drops towards the end of the search, the search will increasingly favor downhill moves only - shifting its focus onto finding as good solutions as possible. This balances intensification and diversification in a logical manner: diversify first, discovering the promising areas of the solution space. When the solution space is sufficiently explored, shift focus onto finding the best solutions within the promising regions.

Since the nominator of the Boltzmann function is dependent upon objective values, the denominator - the temperature - must also have this characteristic somehow. To set the initial temperature T_0 the search starts off with a *warmup phase*, a period of the search lasting for 100 iterations where solutions are accepted with a probability determined by the hyperparameter p_{warmup} (usually set to 1.0). During this phase all the cost differences of the non-improving moves $\Delta E > 0$ is kept track of by the algorithm. When the warmup phase ends, the mean positive delta $\overline{\Delta E}$ is calculated, from which the initial temperature is derived:

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

During the rest of the search, the temperature is exponentially decreased by a *cooling schedule* presented in Crama and Schyns (2003), reaching about zero temperature - and hence about zero acceptance probability - at the end of the search.

Later in this thesis we present DRLMA, a novel move acceptance which we will compare to the SA acceptance criteria.

2.5 Reinforcement Learning

It is common to divide machine learning into three separate fields, namely *supervised learning*, *unsupervised learning* and *reinforcement learning*. Out of these three, supervised

learning is by far the largest area of research, contributing huge advances within the fields of image processing and natural language processing. In supervised learning a model is trained to discover underlying patterns in a certain data distribution given both a set of samples X and their corresponding target variables Y. The overall goal is the maximize the prediction capability of the model. Given an unseen set of data from the same distribution, the desire is to have the predicted targets Y_{pred} to resemble the true targets Y_{true} as much as possible. For instance, X might be pictures of animals and Y the corresponding type of animal. The supervised learning task is thus *animal recognition* - to be able to tell the type of animal given a picture of it. Similarly, unsupervised learning is also concerned with a data set X from some underlying distribution, but unlike supervised learning, it doesn't utilize the target variables Y. The objective of unsupervised learning is rather to gain some insight into the nature of the data distribution itself, and some typical unsupervised tasks are dimensionality reduction, clustering, and anomaly detection.

Reinforcement learning, which is the focus of this thesis, is quite different from the two other fields of machine learning. The most important distinction is the absence of a predefined data set. Reinforcement learning is concerned with the learner and decision maker one the one hand, denoted *agent*, and the environment which the agent interacts with on the other hand. For each action the agents makes, it is given a scalar value, denoted *reward*, from the environment. The reward can be interpreted as the environment's response to the agent's action, and it can be both positive and negative. It gives the agent an indication of the quality of the action. The agent will use this feedback to try to learn which actions are favorable in a given situation. The agent's objective is to maximize the received reward in the long run.

This chapter will firstly give a brief introduction to reinforcement learning and its central concepts. Then, the concept of Deep Reinforcement Learning (DRL) will be introduced, along with one of its most successful approaches in recent times.

2.5.1 Introduction to Reinforcement Learning

Reinforcement Learning (RL) is learning how to map situations to actions so that some numerical reward is maximized. The agent and the environment interact continually in a circular fashion. The agent selects an action based on what it currently observes, somehow utilizing the knowledge it has learnt. The environment executes the action, transitions the agent into a new situation, and provides the agent with an immediate reward. This circular dynamic is displayed in Figure 2.2.

More precisely, the agent is given a *state* $s_t \in \mathcal{S}$ from the environment at time step t, where $\mathcal{S}$ is the set of all possible states. A state is a representation of the environment as things currently is - every bit of information the agent should know to make a decision. Given a state, the agent chooses an action a_t from its own action space $\mathcal{A}(s_t)$. Both the next state s_{t+1} and the reward r_{t+1} is determined by the *dynamics* of the environment. In cases where the number of possible states, actions and rewards are finite, the dynamics can be mathematically formulated as the conditional state-transition probability function

$$p(s', r|s, a) = P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a) \tag{2.2}$$

for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$ and $a \in \mathcal{A}(s)$. Being a conditional probability distribution, it has the following property:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s). \tag{2.3}$$

This means that the agent has a certain probability of ending up in state s' given that it currently is in state s and performs the action a. In many reinforcement learning scenarios, this probability is zero for all but a few states. It can be useful to think of the next state and reward pair as being sampled from this probability function. One important aspect of the environments dynamics is whether it is stationary or dynamic. In a stationary environment where the transition probabilities and reward function stays constant, learning is much easier since any knowledge the agent acquires will also be true in the later parts of the learning process. In our experiments the environment is stationary, but we will briefly discuss a potential improvement upon or own contribution toward the end of this thesis where the environment becomes dynamic.

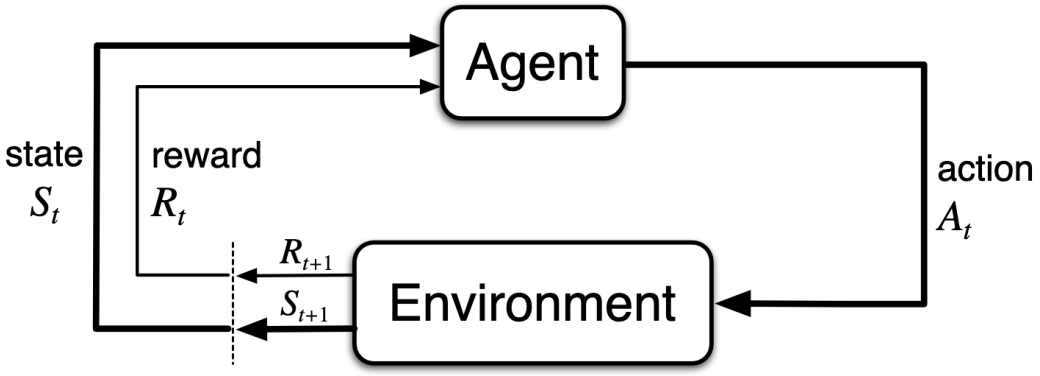


Figure 2.2: The agent-environment interaction cycle. At state S_t the agent selects action A_t. The environment transitions the agent into the new state S_{t+1} along with the immediate reward R_{t+1}. Image from Sutton and Barto (2018).

Another important aspect of the environments dynamics is that it is often unknown. The only way of gaining knowledge about the quality of an action given a state is by having an agent explore - choose some action, receive some reward, and learn from the result. Recall that the agent's objective is to maximize the long-term reward signal. This long-term reward can formally be defined as the *discounted return*, commonly referred to as just the return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.4}$$

where $\gamma \in [0, 1]$ is the *discount rate* (Sutton and Barto, 2018). The discount rate balances how much the agent values immediate rewards versus rewards far into the future. Note that when attempting to optimize the return, the agent might not always choose the action with the highest immediate reward - a seemingly bad action might be the optimal choice in the long run.

A central concept in RL is the *value function*, which gives the expected return for any state or state-action pair. This is a theoretical concept the agent has no direct access to. However, some RL-approaches maintains a value function estimate. In such cases the estimate is continually updated as the agent explores the environment and receives more knowledge about its behaviour. The value function goes hand-in-hand with the agents *policy*, meaning its behaviour. Formally, the policy can be defined as a probability distribution over all possible actions conditioned on the current state, denoted $\pi(a_t|s_t)$. It can be thought of as a mapping from states to actions. The policy is thus highly interpretable - it explicitly states the agents behaviour. The state-value function v is theoretically defined with respect to a certain policy π:

$$v_\pi(s) = \mathbb{E}_\pi(G_t|S_t = s) \tag{2.5}$$

that is, the expected return from state s if the agent follows the policy π at time step t. Instead of explicitly stating which action the agent should take, the value function states how preferable a certain state is.

Using these concepts, the field of RL can be roughly divided into the following approaches (Sutton and Barto, 2018):

- Value-based: Has a learnt value function, from which the policy is implicitly derived
- Policy-based: Learns the policy directly
- Actor-critic: Learns both the policy and the value function

Value-based methods aims to estimate the true value function - the expected return for all states or state-action pairs. The goal in RL is usually to improve upon an already known policy, and possibly even to find the *optimal policy* - the policy that maximizes the expected return of all states. The idea of value-based methods is to use the value function to search for good policies. A common approach is to estimate the state-value function $V_\pi(s)$ for a given policy π. The value function estimate only tells you how good a particular state is - it doesn't clearly communicate the best action given a state. Therefore, many value-based RL approaches revolves around estimating the action-value function $q_\pi(s,a) = \mathbb{E}_\pi(G_t|S_t = s, A_t = a)$ instead. Theoretically it is equivalent to the value function, and it is immediately clear which action that will likely yield the highest return. A simple and yet effective way obtaining an action-value estimate Q_π is by applying the recursive Bellman equation (Bellman, 1954):

$$Q_\pi(s,a) = \mathbb{E}_\pi[r_{t+1} + \gamma Q_\pi(s_{t+1}, \pi(s_{t+1}))] \tag{2.6}$$

The idea is to repeatedly explore the environment and continually update the estimate for the encountered state-action pairs by combining the immediate reward with a one-step look-ahead. If the right conditions are met it can be theoretically shown that this procedure will converge to the action-value function of the optimal policy. One of these conditions is that all states must have a non-zero probability of being visited, which means that *exploration* must be a part of the trajectory sampling process in some way. Examples of value-based methods that estimate the action-value function to obtain good policies are SARSA and Q-learning.

While value-based methods aims to find the optimal policy through value-function estimates, *policy-based methods* aims to directly learn the optimal policy. One common approach is *policy gradient methods*, which parameterize the policy by using a parameter vector $\boldsymbol{\theta} \in \mathbb{R}^d$. The policy, denoted $\pi(a|s,\boldsymbol{\theta})$, can be seen as a function that inputs the current state and the parameter vector, and outputs the probability distribution over all possible actions. The most straightforward approach is to represent the policy parameters as a one-dimensional vector, which corresponds to linear approximation. The parameters may also represent the weights of a deep neural network, a topic we will return to shortly. The *objective function* of policy gradient methods is the expected return from the start state following the current policy: $J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0)$. Recall that this quantity is what we want to maximize. The *policy gradient theorem* states that the gradient of this quantity with respect to the parameters, $\nabla J(\boldsymbol{\theta})$, can be calculated using some theoretical quantities including the true value function. In theory then, the parameters

can be updated using gradient accent:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta} + \alpha \nabla J(\boldsymbol{\theta}) \tag{2.7}$$

where α is the learning rate - the degree of change applied to the policy parameters. Since the true value function of the policy is unknown, the gradient $\nabla J(\boldsymbol{\theta})$ needs to be approximated. One possibility is to replace the value function term with the observed return G_t, which yields an unbiased estimate of the gradient. This is the idea behind the straightforward algorithm known as REINFORCE (Williams, 1992). However, return-based estimates are often noisy and can result in drastic updates to the parameters. This is the main reason why these policy gradient methods have a high variance and suffer from slow convergences. Luckily, there exits techniques that aim to reduce this variance. One approach is known as baseline subtraction from the estimated return, which can stabilize learning. Another approach is known as *actor-critic methods*.

In actor-critic methods both a value function and a policy is learnt, in hopes of combining the advantages of both value-based methods and policy-based methods. The actor corresponds to the policy and is responsible for selecting the actions, and the critic corresponds to the estimated value function. By having a value function estimate it's possible to estimate the *advantage* of taking some action in some particular state. The advantage, as a theoretical quantity, is the difference between the expected return of taking some action and the expected return of following the current policy given a state. This provides a way of assessing the actions in question: a positive advantage means that the action is preferable compared to following the current policy. The policy can then be updated to favor actions that has a high advantage, which can lead to faster and more stable learning. Replacing the empirical return with the advantage estimate reduces the variance at the cost of introducing some bias (Schulman et al., 2018). In the recent years, several actor-critic approaches is has shown state-of-the-art-like performance on a wide range of reinforcement learning tasks, especially in combination with the recent developments of deep learning.

2.5.2 Deep Reinforcement Learning

If the state and action space for a reinforcement learning problem is limited in size, it is sufficient to represent the value function or the policy as a table that maps state-action pairs to values or probabilities. Such approaches, called *tabular solution methods*, updates the values of the state-action pairs the agent visits based on the rewards received and

the observed state transition that follows. However, many real-life problems are often very complex and consist of a high-dimensional state space. The state space of such problems is often huge and may even be infinite. For these kinds of problems we simply cannot hope to learn the optimal policy using tabular methods, because the state space cannot be sufficiently explored. There is also the practical issue of fitting the entire state table into the computer memory. In these situations approximate solution methods are in order, where the goal is to find sub-optimal but good enough policies.

DRL is a subfield of machine learning that combines deep learning with reinforcement learning. Deep learning is used to scale the classical RL methods, making them applicable to problems earlier thought to be intractable. The greatest achievement within the field of DRL to date is arguably when Google DeepMind's deep learning-based RL-program AlphaGo was able to beat the sitting world champion Lee Sedol in the game of Go with a 4-1 score in March 2016. The game of Go is often viewed within the artificial intelligence community as the most challenging of classical games to master due to its enormous state and action space, and before the showdown this achievement was thought to lie at least a decade ahead (Silver et al., 2016). This alone truly demonstrates the potential of the area of research that is DRL. Besides that, DRL also has practical applications in fields of robotics, healthcare, transportation, video games and finance.

Deep neural networks (DNN) are essentially function approximators very capable of learning to capture complex relationships between a high-dimensional raw input space and the output space. There are several benefits to this that makes them appropriate in the reinforcement learning setting. The use of DNNs enables so called *end-to-end learning*, which means that the network input is the raw, unperturbed state space and the output is either the action to take or the value of the state or the state-action pair. Thus, the DNN approximates either the value function or the policy directly, eliminating the need for manual feature extraction, which in practise can be a tedious and time-consuming task. For example, convolutional neural networks (CNNs) can be integrated into the DNN, making it capable of handling raw, high-resolution imagery as input.

In contrast to tabular methods, DNNs don't suffer from the explosion of the dimensionality of the state space which typically happens for complex RL problems. On the contrary, DNNs are known to generalize, enabling the agent to behave reasonably when facing previously unseen states. This is due to the powerful concept of *representation learning*: the neural network learns to represent the high-dimensional raw input as low-dimensional features without losing the relevant information. These features can be generalized across similar states, which speeds up learning and removes the need for

exploring the entire state space. This assumes the smoothness assumption: if some state action pair (S, A) was a good choice, and another state S' is similar to S, then chances are that also (S', A) is good. For many RL problems, this is a reasonable assumption to make.

The field of DRL really took off when a novel algorithm was shown to display professional human-level performance on a range of Atari 2600 video games, learning to play based on raw image pixel data and the game score (Mnih et al., 2015). Their algorithmic contribution, know as Deep Q-Network (DQN), combines a variant of Q-learning with the deep convolution neural network architecture. Since the success of DQN, a number of algorithmic improvements has been proposed. One such example is Double Deep Q-Network (DDQN), which uses two separate networks in training: one for action selection and one for action value estimation. This decouples action selection from action value evaluation, in turn mitigating the tendency DQN has of overestimating action values, leading to faster learning and an overall better performance on a range of problems (van Hasselt et al., 2015).

Both DQN and DDQN is value-based methods. Although powerful, they suffer from some shortcomings, the biggest one being related to the fact that value-based methods estimates the future reward of state-action pairs, and have implicit policies thereof. When updating the value estimates, the policy might change drastically, which can lead to issues like instability and oscillation in the Q-values, and ultimately to poor performance. Also, value-based methods aren't able to handle stochastic policies, and would fail in a simple rock-paper-scissors environment. This, however, is no problem for policy-based methods, which handles stochastic policies perfectly fine. However, policy-based methods tend to be both data inefficient compared to their value-based counterparts and may suffer from converging to severe local minima - that is, unsatisfactory policies.

Proximal Policy Optimization

With policy-based methods it can be challenging to update the policy in a stable way, especially if the state space is vast and generally unstable (Wang et al., 2020). As mentioned earlier, a naive weight update using some gradient estimate might lead to large changes in behaviour, which can result in disastrous policies. To combat this, an actor-critic method called Trust Region Policy Optimization (TRPO) was developed, which restricts the degree the policy is allowed to change for each weight update. This is achieved by

introducing a constraint term into the objective function. A downside to this is that solving constrained optimization problems is rather computationally expensive, as it requires a quadratic approximation to the constraint.

Its more natural to utilize a close relative of TRPO called Proximal Policy Optimization (PPO), introduced by Schulman et al. (2017). In PPO the constraint term is replaced by a *clipped surrogate objective*, which can be solved using only first-order optimization, like gradient accent. This objective *pessimistically* ensures that the new policy doesn't deviate too far from the old policy. That is, even though there is indications that the new policy is improved, only a small policy update is performed. This can result in slower learning, but typically it rather leads to more stable and consistent learning, which often is the most desirable thing in reinforcement learning. Also, the clipped surrogate objective allows for using several epochs, i.e. gradient updates per data sample. This increases data efficiency, essentially increasing the size of the training set without having to increase the amount of interaction with the environment. Thus, the policy can be gradually refined in a controlled manner, essentially using the available information in the data to its fullest extent.

When it was launched in 2017, PPO displayed state-of-the-art performance on typical RL benchmarks like Atari games compared to other policy-gradient methods. In comparison to TRPO, PPO is simpler to implement, less computationally expensive and requires less tuning of hyperparameters. Due to its simplistic and versatile nature, this algorithm is a natural first choice when faced with a reinforcement learning task, even today. We use PPO as the RL backbone through this entire thesis.

2.6 Related Work

2.6.1 Metaheuristics and Reinforcement Learning

Although the idea of incorporating reinforcement learning into both metaheuristic and hyperheuristic framework is relatively new, a fair amount of research has been done on the topic. Zhang and Dietterich (1995) was one of the first papers which combined reinforcement learning and metaheuristics, according to Wauters et al. (2013). This methods learns the value function on a small set of instances of the job-shop scheduling problem, using domain-specific constructive heuristics. These heuristics make up the action space of the RL model. This approach was shown to outperform the leading

heuristic algorithm for this task at the time, namely an iterative repair method based on SA, on multiple new instances of the same problem.

The advent of DRL has inspired researchers to incorporate typical DRL techniques into heuristic solution methods. Several construction-based hyperheuristic methods has been proposed, such as Kool et al. (2019), Nazari et al. (2018) and Zhang et al. (2022). Although outperforming other simpler construction heuristics, the solutions obtained lack the quality that perturbative metaheuristics are able to produce. Lu et al. (2020) uses a Deep RL agent for the selection of low-level heuristics in a perturbative hyperheuristic framework. A limitation of all of these mentioned works is that they somehow rely on problem-dependent information, which limits their applicable scope to a single problem or a class of similar problems.

As far as we are aware, Kallestad et al. (2023) was the first work to propose a *general* hyperheuristic framework which solely relays on a Deep-RL agent for heuristic selection. Their contribution, called Deep Reinforcement Learning Hyperheuristic (DRLH), uses at each time step a state representation to produce a probability distribution over the low-level heuristics, from which the heuristic choice is sampled. As opposed to ALNS which uses segment-locked probabilities, DRLH is able to take advantage of the available information at every single time step, producing drastically varying probabilities from one iteration to the next. In their work they show that DRLH manages to significantly outperform ALNS on a wide set of combinatorial optimization problems with a fixed set of hyperparameters.

2.6.2 Move Acceptance

In the original version of ALNS, Ropke and Pisinger (2006) proposed using the SA acceptance criteria - without putting a lot of emphasis on the reasoning behind this choice. In later years there has been several attempts on comparing the most prominent acceptance criteria in literature. Jackson et al. (2018) compared several different acceptance criteria backed by the a hyperheuristic known as HyFlex (Hyde et al., 2010) on nine different combinatorial optimization problems. They find that although the SA acceptance criteria isn't outperforming the other acceptance criteria on any of the specific problems, it consistently performs quite well across all the nine problems - coming out as the leading move acceptance *in general*. Perhaps more relevant to our own research, in their comparison of different move acceptances within the ALNS framework, Santini et al. (2018)

found that the move acceptance known as Record-to-Record Travel (RRT) was consistently undominated by the others - a group of acceptance criteria which included Hill climbing, Great deluge and the SA acceptance criteria. The findings of Hemmati and Hvattum (2017) confirm the superiority of the RRT acceptance criteria compared to the SA acceptance criteria on the Pickup and Delivery Problem.

RRT, first proposed by Dueck (1993), is a deterministic criteria that accepts uphill moves within a certain threshold T of the best seen solution. There are several implementations of this criteria in literature, so here we recite the one used by Hemmati and Hvattum (2017). More formally then, the worse solution l'_t is accepted if $f(l'_t) < f(l_t^{best}) + T$, where $T = 0.2(\frac{N-t}{N})f(l_t^{best})$. Here, N is the total number of search iterations (perturbations) and t the current iteration number. Observe that the threshold T decreases linearly in the iteration number, shifting the focus gradually from diversification to intensification as the search progresses - an idea shared with the SA acceptance criteria. In our experimentation, we will combine ALNS with both the SA acceptance criteria and the RRT acceptance criteria, and compare their performances with our own contributions.

2.6.3 Move Acceptance and Reinforcement Learning

As far as we are aware, very little research has been conducted on the topic of incorporating techniques of RL into the move acceptance . Wauters et al. (2013) is the only work we could find which leverages RL in the move acceptance. Their contribution improves upon the Iteration Limited Threshold Acceptance (ILTA), which accepts small uphill moves after a certain number of non-improving candidates. These uphill moves are limited by a range R, a certain objective difference between the candidate l'_t and the current best known solution l_t^{best}. In ILTA, R is a hyperparameter and stays constant throughout the search. Their contribution, called LA-ILTA, uses a RL-agent to select the value of R from a predefined set of reasonable values. The search process is divided into 10 periods, and at the start of each period the agent selects the value of R which will remain constant for that period. This makes the search process more adaptive and removes the need for carefully finetuning the hyperparameter. Their approach was able to outperform regular ILTA on two different combinatorial optimization problems, namely the Patient Admission Scheduling Problem and the Edge Matching Puzzle Problem. An immediate weakness with this approach is that the set of R-values must be tailored to the particular problem type and size, requiring some knowledge about the problem nature in advance along with further tuning of the selected values.

In our opinion, all the move acceptances we have discussed so far has a common weakness, namely that they work quite elementary on a technical level. The criteria we have touched upon are concerned with some objective difference threshold, defined either deterministically (hard threshold) or probabilistically (soft threshold), whose magnitudes are decreased throughout the search. Our main critique is that only a marginal fraction of the search information available is actually utilized. Furthermore, neither of them take advantage of the learning capabilities and representation power of Deep RL.

Let us now introduce the concept of leaving the acceptance decision up to a Deep RL-agent. The authors of this thesis experimented with this concept prior to this thesis, as part of a graduate-level project course at the Department of Informatics at the University of Bergen in the Spring of 2022. In a proof-of-concept like manner they showed that an RL-based move acceptance was able to outperform both the SA acceptance criteria and the random acceptance criteria on several sizes of the CVRP. In their experimentation, ALNS was used as the hyperheuristic. Their results were presented in an unpublished project report. As of today we are unable to locate any published research that resembles this approach.

In this thesis we further develop the contributions of the aforementioned project into a general move acceptance framework which we call Deep Reinforcement Learning Move Acceptance (DRLMA). Moreover, we show that the findings of Kallestad et al. (2023) can be further improved upon by unifying their contribution, DRLH, with DRLMA. We display the effectiveness of DRLMA on two different combinatorial optimization problems, namely the Capacitated Vehicle Routing Problem (CVRP) and the Parallel Job Scheduling Problem (PJSP).

Chapter 3

Problem Sets

3.1 Capacitated Vehicle Routing Problem (CVRP)

The Capacitated Vehicle Routing Problem is certainly one of the most frequently encountered routing problems in the literature. The problem deals with efficient distribution of orders from a central depot to a set of geographically dispersed customers using a fleet of vehicles. The objective of CVRP may vary, but for our purposes it is to minimize the total distance travelled while at the same time satisfying a set of constraints.

More specifically, N orders must be delivered to their associated customers through the use of a vehicle. A vehicle must begin its journey at central depot, deliver some orders, and then return to the depot. Such a journey is referred to as a *tour*. The vehicle has a maximum capacity, and each order i has an associated weight W_i. When departing from the depot the total weight of the set of orders handled in the current tour cannot exceed the vehicle's maximum capacity. Typically then, several tours must be made in order to handle all the orders. The objective is to construct a set of tours that minimize the total distance travelled. Note that there is no incentive to minimize the number of tours made - this number may be as high as N. Note also that in this version of the CVRP, the size of the fleet of vehicles isn't relevant. All M tours may be handled in parallel by M vehicles, or they may be handled consecutively by a single vehicle.

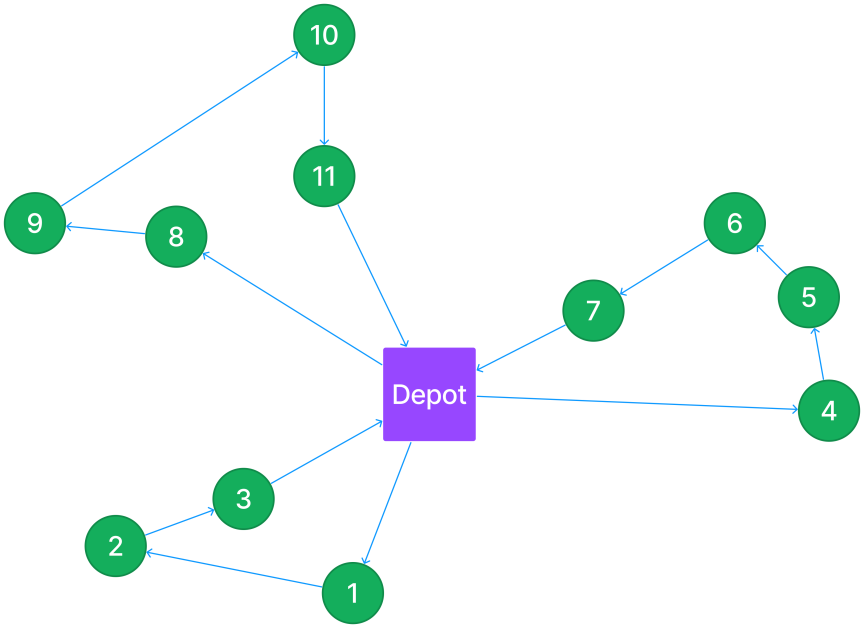


Figure 3.1: Illustration of the CVRP.

3.2 Parallel Job Scheduling Problem (PJSP)

The Parallel Job Scheduling Problem (PJSP) is a combinatorial optimization problem that involves scheduling a set of jobs onto a set of parallel machines. There exists several variants of this problem, and in this variant each machine operates at a different processing speed, and each job has an associated deadline or due time. The objective is to minimize the total delay of the jobs, which is the sum of the lateness of each job.

More formally, the PJSP consists of a set n jobs, denoted $J = \{J_1, J_2, ..., J_n\}$, and a set of m parallel machines, denoted $M = \{M_1, M_2, ..., M_m\}$. Since the machines operate at different processing speeds, each job J_i has a processing time $T_{i,j}$ associated with each machine M_j, representing the amount of time required to complete job J_i on machine M_j. Each job J_i must be assigned a machine and is given a start time S_i. The lateness of job J_i is given by $l_i = \max(0, S_i + T_{i,j} - d_i)$, where d_i is the the deadline of job J_i. That is, the lateness of a job is the difference in its completion time and its deadline, and it cannot be negative. Each machine is assigned a sequence of jobs such that all jobs are handled exactly once, and the objective to find a job sequence distribution such that the total job lateness $\sum_{i=0}^{n} l_i$ is minimized.

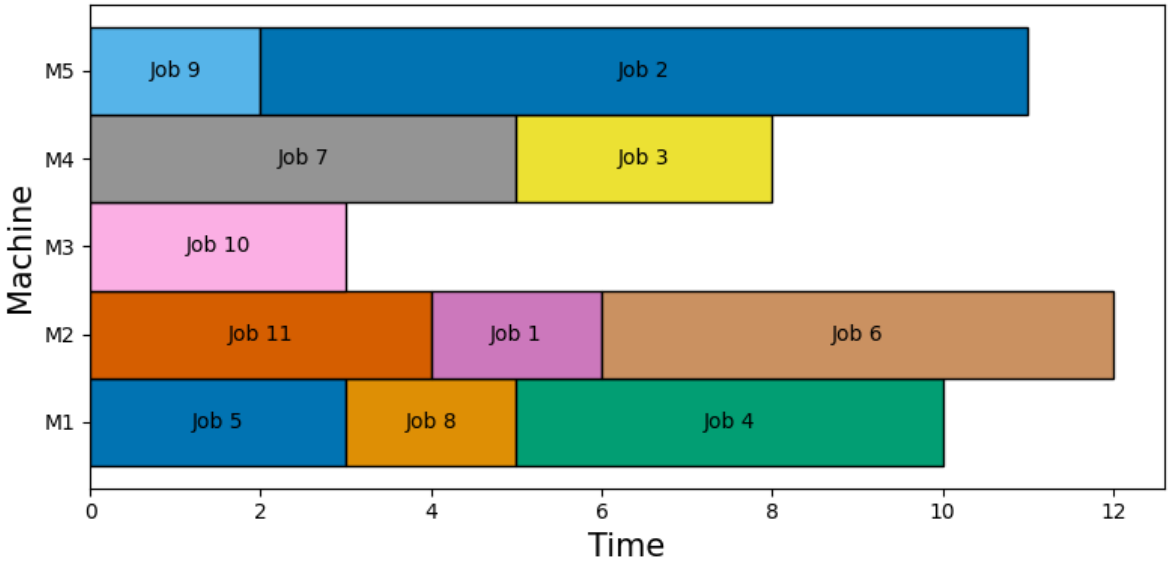


Figure 3.2: Illustration of the PJSP.

Chapter 4

DRLMA

In this chapter we present the novel contribution of this thesis, the **D**eep **R**L **M**ove **A**cceptance (**DRLMA**) - a move acceptance based on Deep Reinforcement Learning. This is a standalone move acceptance framework, meaning it can be combined with any heuristic selection strategy. We show this by inserting it into two different hyperheuristic contexts, namely ALNS and DRLH. In settings where both solution methods are brought up, we use the term *DRLMA* to refer to the ALNS setting and *DRLH+DRLMA* to refer to the DRLH setting. Otherwise, *DRLMA* will refer to just the move acceptance framework itself. The emerging solution method in which DRLMA is embedded is presented in Algorithm 1

4.1 The Hyperheuristic Setup

Before diving into the details surrounding DRLMA, we first present details regarding our hyperheuristic setup that remains constant throughout all experiments. This includes the details of the heuristics we leverage, including the way they are constructed. Further more, we present the stopping criteria, the solution representations and the initial solutions that we use in our experiments. Lastly, we introduce the concept of including Deep RL into the hyperheuristic setting.

4.1.1 Heuristics

In all of the solution methods used in our experimentation we use the same set of heuristics H. This set consists of two classes of heuristics: the first class, which makes up most of the heuristics $h \in H$, is a combination of destroy and repair heuristics which is presented in Tables 4.1 and 4.2 respectively. The second class consists of one intensifying heuristic c which doesn't possess the destroy-repair combination property. All the heuristics are problem independent, meaning that they can be implemented and applied to most combinatorial optimization problems. The heuristic set construction process is described in Algorithm 2.

Algorithm 1: Hyperheuristic framework with Deep RL Move Acceptance

Input: Move Acceptance Policy θ, Hyperheuristic $hh()$

Output: l_{best}

Function `Hyperheuristic()`**:**

Generate initial solution l with objective $f(l)$ **(see section 4.1.2)**

H = Generate_Heuristics() **(see Algorithm 2)**

$l_{best} = l$

repeat

choose $h \in H$ based on hyperheuristic $hh()$

$l' = h(l)$

if $f(l') < f(l_{best})$ **then**

$l_{best} = l'$

end if

if *accept*(s_t, θ) ***(see section 4.2)*** **then**

$l = l'$

end if

until *stop-criteria met*;

Algorithm 2: Generating the heuristic set H

Input: $\mathcal{D}$, $\mathcal{R}$, $\mathcal{C}$

Output: H

Function `Generate_Heuristics()`:

```
H ← {};
foreach destroy heuristic d ∈ D do
    foreach repair heuristic r ∈ R do
        Combine d and r into a heuristic h;
        H ← H ∪ h;
    end foreach
end foreach
foreach additional heuristic c ∈ C do
    H ← H ∪ c;
end foreach
```

In our case, the cardinality of the heuristic sets is as following: $|\mathcal{D}| = 7$, $|\mathcal{R}| = 4$ and $|\mathcal{C}| = 1$. Thus, the final set of heuristics H has a size of $|H| = 29$ (7 destroys $\times$ 4 repairs + 1 additional). At each iteration of the search process, one of these 29 heuristics $h \in H$ is selected by the hyperheurstic and applied to the incumbent solution l with a cost of $f(l)$, producing a candidate solution l' with a cost of $f(l')$. The following subsections describes the destroy, repair and additional heuristics in closer detail.

Destroy Heuristics $\mathcal{D}$

The destroy heuristics set $\mathcal{D}$ is presented in Table 4.1. The first five heuristics removes elements from the solution at random. They differ in size - more specifically in the amount of elements to remove, a number which is randomly sampled from a predefined range. These five heuristics have a strong diversifying effect. On the other hand the next heuristic, called *Destroy_largest_*$\mathcal{D}$, provides a more intensifying effect, using a concept we refer to as the *deviation* $\mathcal{D}$. In this context, we define the deviation $\mathcal{D}_i$ as the cost difference of the solution with and without element i present in the solution. This heuristic thus removes the n elements with the largest $\mathcal{D}_i$, where n is randomly sampled from the range $[2, 5]$. The last heuristic, *Destroy_*τ, removes a randomly selected sequence of n back-to-back elements in the solution, where n is sampled in the same way as with *Destroy_largest_*$\mathcal{D}$.

Name	Description
Random_destroy_XS	Removes between 2-5 random elements
Random_destroy_S	Removes between 5-10 random elements
Random_destroy_M	Removes between 10-20 random elements
Random_destroy_L	Removes between 20-30 random elements
Random_destroy_XL	Removes between 30-40 random elements
*Destroy_largest_*$\mathcal{D}$	Removes between 2-5 elements with the largest $\mathcal{D}_i$
*Destroy_*τ	Removes a random segment of 2-5 successive elements

Table 4.1: The destroy heuristics.

Repair Heuristics $\mathcal{R}$

The repair heuristics set $\mathcal{R}$ is presented in Table 4.2. We use four repair operators, whose purpose it is to "repair" the "destroyed" solution l' by relocating the removed elements into befitting locations of l'. Firstly, its worth mentioning that the insertion order is random if not specified otherwise. *Repair_greedy* inserts the removed elements in the location within the solution that minimizes the total cost. The immediate problem with this heuristic is that a seemingly optimal insertion of one element might result in poor insertions of the consecutive elements. The *Repair_beam_search* heuristic attempts to mitigate this problem by conducting a beam search with beam width 10 when inserting each element. This beam search maintains track of the 10 best insertion combinations seen so far. When a new element is to be inserted, the 5 best solutions for each beam is kept, creating 50 candidate solutions. Of these, the 10 best candidates is selected as the next-generation beam, and the process is repeated until all elements is inserted. The *Repair_by_variance* heuristic calculates the variance in cost between the 10 best insertion locations for each element. A large variance indicates that some locations are far better or worse than others, which again is arguing for a certain "urgency" of inserting the particular element. The elements is inserted back into the solution in their best possible position similar to *Repair_greedy*, but in the order of highest variance first. Lastly, the *Repair_first* heuristic inserts the elements in the first feasible position found within the solution. A certain degree of randomness is introduced by this heuristic by randomly selecting the order of solution sub-parts to attempt insertion at.

Name	Description
Repair_greedy	Inserts elements in their best possible location
Repair_beam_search	Inserts elements in their best location using beam search
Repair_by_variance	Inserts elements in their best possible location, where insertion order is based on variance
Repair_first	Inserts elements randomly at first feasible location

Table 4.2: The repair heuristics.

Additional Heuristics $\mathcal{C}$

In addition to the destroy-repair heuristics typically utilized by LNS and ALNS, we use one additional heuristic in our experiments which doesn't possess this characteristic, namely the *Find_single_best*. This heuristic focuses entirely on intensification by producing the best possible solution modifying only a single element. More specifically, every element is attempted removed and greedily reinserted, and the modified solution l' with the lowest cost $f(l')$ is finally yielded.

4.1.2 Solution Representation and Initial Solution

To represent a solution to a problem we use permutations of orders or jobs, where each vehicle or machine is assigned such a permutation. From a vehicle's perspective the permutation represents its route. From a machine's perspective it represents the sequence of jobs to handle. All of the orders/jobs must occur in the permutations exactly once.

The initial solutions to the problem instances are constructed in a deterministic and rather inelegant fashion. For the CVRP consisting of n orders, the initial solution consists of n tours, each tour delivering a single order. For the PJSP consisting of n jobs, all the machines but the last one is assigned zero jobs and the last machine is assigned all the jobs. Quite obviously, these solutions are far from optimal, but they provide a fair and equal starting point for all the solution methods in our experiments.

4.1.3 Stopping Condition

Recall that the stopping condition is usually either defined as a time limit or as a limit on the number of permutations - or search iterations. In this thesis we use the latter stopping condition, as was used by Kallestad et al. (2023). For all the solution methods in our experiments, the search terminates after 1000 search iterations.

4.1.4 Deep Reinforcement Learning in the Hyperheuristic setting

Reinforcement learning is concerned with having an agent interact with its environment while simultaneously attempting to optimize the policy π. The policy is typically represented indirectly through an action-value function $Q(s, a)$ or directly through a policy function $\pi(s)$. In the latter approach, the policy can be parameterized through a set of parameters $\theta \in \mathcal{R}^d$ in the following way

$$\pi(a|s, \theta) = Pr\{A_t = a|S_t = t, \theta_t = \theta\} \tag{4.1}$$

where at each time step t, the state S_t is the information available to the agent, A_t is the action it selects and θ_t is the current policy parameters.

In our approach we utilize a class of non-linear function approximators called Multi-Layered Perceptron (MLP) to represent the policy π, which is one type of DNNs. The weights that make up the layers of the MLP represents the policy parameters θ. Between the layers we employ an *activation function* called $ReLU$ which is widely regarded as the golden standard of activation functions. As mentioned the parameters are trained according to the actor-critic method known as PPO (Schulman et al. (2017)). The training process is described in closer detail in Algorithm 3.

Throughout this thesis we train the agents on several different *optimization settings*. In this context, an optimization setting is referring to a specific configuration of the following parameters: 1) the optimization problem being solved, 2) the size of the problem. For each optimization setting explored in this thesis, a new MLP is initialized and trained, as shown in Algorithm 3. Within such an optimization setting, the developed policy should be able to generalize - that is, it should be able to perform well across different problem instances. As such, the agent is trained on a set of problem instances called the *training set*, each instance varying slightly from the next in attributes not related to the size of the problem. Each instance is optimized until the stopping criteria is met, which marks the end of the RL episode. Then, the weights of the MLP is updated using the experience gathered from the episode. This process is repeated for the entire training set. The overall goal is to find an as good approximation of the optimal policy π^* as possible. However, the agent's performance on the training set is a biased estimate of its performance - it doesn't relay the policy's ability to generalize within this optimization setting. Therefore, the trained policies are used to optimize a smaller set of unseen

instances called the *test set*. The test performances can then be used to determine good policies from bad ones.

Algorithm 3: Training the RL agent

Output: $\pi_\theta \approx \pi^*$ the optimal policy

Function `Training_loop()`**:**

Initialize policy weights θ_0 in a uniform random fashion;

for $e \leftarrow 1$ **to** *episodes* **do**

Receive initial state s_1 using initial solution;

for $t \leftarrow 1$ **to** *steps* **do**

Sample and perform action $a_t \in \mathcal{A}$ according to $\pi(a|s, \theta_{e-1})$;

Receive reward r_t and next state $s_{t+1} \in \mathcal{S}$ from the environment;

end for

Update policy $\theta_e \leftarrow \text{PPO}(\theta_{e-1})$ as described by Schulman et al. (2017)

end for

When solving any problem using RL there are two central design choices one has to make regardless of the RL algorithm used, namely the *state representation* and the *reward function*. Getting these design choices right is essential for learning good policies and by extension finding good solution methods to the optimization problems. There is also the generalization aspect of these two design choices: in our thesis, the state representation and reward function is invariant to the optimization setting, indicating that our approach is able to generalize across problems. The state representation should contain all the information considered relevant to the agent's decision in the search process, without containing any problem-specific details. The reward function is the very thing that the agent attempts to maximize, so it should be aligned with the overall goal of producing an as good solution method as possible. This alignment is no easy task, and seemingly reasonable reward functions might be exploitable by the agent, a phenomenon known as *reward hacking* (Amodei et al., 2016). This phenomenon can be characterized as the agent having learnt a policy that receives high rewards while at the same time performing terrible at the task, even often coupled with nonsensical behaviour. We will discuss these two design choices in the context of our solution method in the following section.

4.2 Deep RL Move Acceptance (DRLMA)

It is our opinion that the other common move acceptances in literature are quite elementary on a technical level, in the sense that their decisions are based on a small

amount of available information regarding the search. For instance, the SA acceptance criteria is based on two parameters: 1) how much worse the proposed solution is and 2) the temperature, which decreases exponentially as the search approaches the stopping criteria. However, there might exist better move acceptance strategies that depend on even more information about the search process. These strategies might be obtainable automatically through the power of Deep RL instead of manually developed by experts. Furthermore, Deep RL is able to capture complex non-linear relationships between the inputs and outputs, allowing the learnt strategies to be quite sophisticated. Thus, we hope to learn an Deep RL-fueled move acceptance to explore the solution space more effectively than other state-of-the-art move acceptance strategies.

The action space of the RL agent is quite simple, namely whether to accept or reject the candidate solution. Note that the RL part of our move acceptance is only applied in cases where the candidate solution is worse than the incumbent. In these cases, the output of the move acceptance is the acceptance probability, from which the action is sampled. In terms of the policy function described by Eq.(4.1), the policy π can be defined as

$$\pi(\text{accept}|s,\theta) = Pr\{A_t = \text{Accept}|S_t = t, \theta_t = \theta\} \tag{4.2}$$

Using the policy then, the acceptance probability of DRLMA is derived from the following formula:

$$P_{\text{DRLMA}}(\text{accept}) = \begin{cases} 1.0 & f(l') < f(l), \\ 0.0 & \text{if } l' \text{ has been encountered before,} \\ p_{\text{warmup}} & \text{if } \textit{warm-up phase}, \\ \pi(\text{accept}|s,\theta) & \text{otherwise} \end{cases}$$

Note also that from a RL perspective, the heuristic selection scheme is a central part of the *environment* in which the DRLMA agent is trained. Therefore, the move acceptance strategy developed in the ALNS setting might substantially differ from the one developed in the DRLH setting. For this reason, we need to train an agent in both settings.

4.2.1 State Representation

The chosen state representation contains a set of features we believe to be useful in the acceptance choices the agent has to make. These features are problem independent: they

contain information regarding the search itself, and about previous decision made by both the heuristic selector and the move acceptance. Thus, our state representation can be applied when solving many different combinatorial optimization problems. Table 4.3 displays the state representation of the RL agent.

In general, it is easier to train a MLP if its input features are either standard normally distributed or confined to some small range, such as between 0 and 1. In our case we separate features into two groups, depending if they are suitable for normalization or not. Our state representation contains 8 non-confined features that we find *normalizable*. For this reason, we keep track of the 100000 previously seen states used to derive the feature-level means μ and standard deviations σ. These statistics are then used to normalize the observed state before being passed on to the RL agent.

Name	**Description**
Normalizable features	
cost	Cost of incumbent solution
candidate_cost	Cost of the candidate solution
min_cost	Cost of the best found solution
cost_from_min	Cost difference between incumbent and best found solution
candidate_cost_from_min	Cost difference between candidate and best found solution
ΔE	Cost difference between candidate and incumbent solution
steps_since_improvement	#iterations since last improvement
steps_with_incumbent	#iterations since current incumbent was accepted
Non-normalizable features	
index_step	How far we are in the search (max normalized)
last_action	Previous action made by the RL network
selected_heuristic	The selected heuristic (one-hot encoded)
acceptance_category	Previous move acceptance action category
candidate_solution_ranks	Cost ranks of candidate solution
solution_ranks_diff	Cost ranks difference between candidate and incumbent solution

Table 4.3: State representation of the RL Move Acceptance agent.

The first six features are related to the costs of the incumbent, candidate and best solution, and how they compare to each other. These features, combined with *index_step*, tells the agent how well the search is doing with respect to how far into the search

it currently is. More importantly, it communicates the relative quality of the candidate solution, which undoubtedly is vital in the decision. *steps_since_improvement* and *steps_with_incumbent* is the number of steps since an improvement happened and the number of steps since the current incumbent was accepted, respectively. These two features relays information about the current balance between diversification and intensification, and is useful for the agent to adjust this balance. For instance, if both values grows large the agent might decide that it should escape the local minima.

last_action, *selected_heuristic* and *acceptance_category* all gives the agent information about recent events within the search. *last_action* tells the agent whether the solution was accepted the last time the RL network made the decision. *selected_heuristic* is a one-hot encoded representation relaying which heuristic made the candidate solution. *acceptance_category* indicates in a one-hot encoded fashion which part of the move acceptance that made the decision the previous step, along with the action, although indirectly. There are mainly four move acceptance cases: 1) improvement, implying acceptance, 2) candidate already encountered before, implying rejection, 3) no special case, implying acceptance and lastly 4) no special case, implying rejection. Lastly, *candidate_solution_ranks* and *solution_ranks_diff* utilize the concept of cost ranks among the solutions encountered so far in the search. This ignores the numerical aspect with costs altogether, making it less sensitive to cost variations across different problem instances and shifts the focus entirely onto instance-specific cost rankings. The ranks are scaled between 0 and 1, where being close to 1 implies a good solution. Three different rankings are reported: a solution's ranking all among 1) solutions which has had the "best solution" status, 2) the accepted solutions, and 3) the seen solutions so far into the search. *candidate_solution_ranks* communicates the quality of the candidate solution within the problem instance, while *solution_ranks_diff* compares the rankings of the incumbent and candidate solutions, relaying their rank distance. Overall, these features supplement the cost-related features in conveying the quality of the candidate solution by comparing the candidate to all the solution encountered in the search so far.

4.2.2 Reward function

The reward function design is vital for learning a useful policy. Firstly, its worth mentioning that Kallestad et al. (2023) utilize the ALNS score function (2.1) as reward function in heuristic selection framework called DRLH. This is simply not applicable in this case, since the move acceptance is part of the ALNS score function. It's unclear how the reward function should be designed, simply because it's unclear if the decision resulted in

a favourable outcome post-decision time. The first idea that comes to mind is to look ahead into the next iteration - if something favourable happened here, say the search found a new best solution, then arguably our decision was a good one. We decided to try giving the ALNS score of the next iteration as reward, modified to ignore the move acceptance part of the score calculation, since this directly depends on the agent's next decision. Some initial experimentation with this design resulted in poor performance.

Even though the DRLH reward function is non-applicable in our case, some inspiration can still be drawn from it. Specifically, the DRLH agent is optimized to produce as many improving solutions as possible, with an extra emphasis on discovering new current-best solutions. Given the quality of their results, they show that the solution strategy can be improved when you increase the number of improving moves within the search. Our reward function should thus reflect this idea.

The reward function we propose relaxes the "real time" reward calculation constraint, allowing rewards to be calculated after the episode is done. This allows information regarding the search process in the near future to be considered when assessing an action. More specifically, the current incumbent and best known solution is compared to the incumbents and best known solutions λ iterations into the future, where λ is a tuneable hyperparameter. Our reward function, which we refer to as $R_t^{\lambda \text{ steps ahead}}$, has the following formula structure:

$$R_t^{\lambda \text{ steps ahead}} = \begin{cases} \omega_{\text{best}}, & \text{if } f(l_{t+\lambda}^{best}) < f(l_t^{best}) \\ \omega_{\text{improvement}}, & \text{if } f(l^\dagger) < f(l_t) \\ \omega_{\text{otherwise}}, & \text{if none of the above} \\ \hdashline \omega_{\text{no improvement \& accept}}, & \text{if } f(l_t) \leq f(l^\dagger) \text{ and } accept(l'_t) \\ \hdashline \omega_{\text{n rejections}}, & \text{if } n \text{ solutions were rejected in a row} \end{cases} \tag{4.3}$$

where $f(l^\dagger) = \min_{i \in [t+1 \ldots t+\lambda]} f(l_i)$. Furthermore, ω_{best}, $\omega_{\text{improvement}}$, $\omega_{\text{otherwise}}$, $\omega_{\text{no improvement \& accept}}$ and $\omega_{\text{n rejections}}$ represent scalar values and are tuneable hyperparameters. The reward function consists of three independent components, separated by dotted lines. The first condition that is fulfilled within each component contributes its associated value ω to total reward for time step t. Thus, several components might be invoked at the same time, and if that is the case their values are added together. Although the reward values are parameterized, the first component reflects desirable behaviour which we positively reinforce by having $\omega_{\text{best}}, \omega_{\text{improvement}} \geq 0$. On the other hand the other components describe

potentially unwanted behaviour, which we discourage by having $\omega_{\text{no improvement \& accept}}$, $\omega_{\text{n rejections}} \leq 0$ The scalar values used in our experiments are presented in Table 5.3.

Let's carefully unpack the intuition behind this reward function. First of all, the first component builds on the assumption that an action is favourable if it leads the search to discover a better solution in the near future. This assumption reflects the idea that the move acceptance should lead the search process into promising parts of the solution space. Note that this component ignores the decision itself. Initially this component made up the entire reward function, and experiments (using $\omega_{\text{otherwise}} = 0$) showed that the trained agents tended to accept all uphill moves. For this reason we introduced the second component, whose main purpose it is to discourage the agent from accepting solutions that doesn't lead to an improvement in the near future. Although this works very well, it leads to more "reserved" policies that generally give quite low acceptance probabilities. To combat this reserved behaviour we introduced the third component, which punishes the agent for rejecting n solutions in a row, essentially forcing the agent be less reserved. Though experiments confirm the learnt policies are less reserved, they also doesn't perform as well, and so we ended up putting $\omega_{\text{n rejections}} = 0$ when training both DRLMA and DRLH+DRLMA.

There is also the choice of number of steps to look ahead in time, λ, when calculating these rewards. A too low λ-value encourages moves that the search can quickly improve upon, and this doesn't allow the search enough time to properly diversify. At the same time, any improvements that happens too far into the future of the search (high λ) is so affected noise that the accomplishment can hardly be attributed to the current decision. In our opinion the optimal choice of λ must be determined experimentally.

4.3 DRLH+DRLMA: A comment on Design Choice

The authors of DRLH showed that they were able to improve upon ALNS by letting a Deep RL agent handle the heuristic selection part of the search. Similarly, the authors of this thesis have previously displayed that the ALNS performance might be slightly boosted by replacing the the SA acceptance criteria with a trained Deep RL agent. This section describes the attempt to unify these two solution methods into a novel hyperheuristic, namely the DRLH+DRLMA. More specifically, we will be discussing the details of how the two approaches might be combined and the reasoning behind our design choice.

There are several ways of carrying out this unification in practice, and each such way have some corresponding benefits and challenges. The setup consists of two collaborating RL agents, the *heuristic selector* agent and the *move acceptance* agent. Both agents have previously been trained in a single-agent environment: the heuristic selector agent in DRLH and the DRLMA agent in the ALNS-environment. Thus, each agent's corresponding neural network, which makes up the the policy π itself, might either be *loaded from disk* or *trained again from scratch*, resulting in four different experimental combinations.

On the one hand, loading both the heuristic selector agent and the move acceptance agent might not induce any extra training time, but we believe it to be a bad idea nonetheless. Both agents are trained in single-agent environments where the opposite decision maker is equivalent to that of ALNS. When put together, the environment is radically changed from the perspective of each agent. A RL agent that performs well in one environment might not necessarily perform satisfactory in a different environment with different dynamics - this is a corollary of the *No Free Lunch Theorem (NFLT)* in optimization and machine learning (Wolpert and Macready, 1997). We believe that freezing both policies is a too rigid constraint, and that better joint policies are obtainable by relaxing this constraint.

On the other hand, training both the heuristic selector agent and the move acceptance agent from scratch offers a lot of flexibility and might seem like the best option, but this also carry some severe challenges. Once several agents are allowed to interact in the same environment and update their policies, you enter into the realm of Multi-Agent Reinforcement Learning (MARL), a more recent and actively evolving field of RL research with several challenges that makes it more complex than single-agent RL. For instance, one of the main challenges of MARL is the presence of other learning agents, which induces non-stationary into the environment. As one agent learn and update its own policy, the environment's dynamics changes caused by the other agent's updated policy, making it difficult to converge to an optimal, joint policy. We consider the MARL-based approach to be outside the scope of this thesis, the main focus being the DRLMA itself. We will return to this idea in Chapter 7.

We consider DRLH+DRLMA to be a "golden mean" between the two approaches just described, attempting to mitigate the change-in-environment problem that arises in the first approach while preserving some of the flexibility provided by the MARL approach. We achieve this by training an DRLMA agent within the DRLH framework, freezing the parameters of the DRLH agent. Note that even though two RL agents are employed

at once, it is not regarded as MARL since only the DRLMA agent is being trained. The DRLH agent is simply regarded as part of the stationary environment. Furthermore, this approach also emphasizes the hyperheuristic-independence of DRLMA. The DRLMA agent can simply be inserted into any hyperheuristic setting, and be trained to increase the performance - see Chapter 6 performance details.

Chapter 5

Experimental Setup

5.1 Experimental Environment

The experiments conducted in this thesis was performed on a desktop computer running a Windows 10 Operating System, possessing an AMD Ryzen 7 5800X processor and 32 GBs of RAM. Our experiments are implemented in a Python 3.10.6 environment, and Table 5.1 displays the Python packages and versions we used.

Package Name	**Version**
Numpy	1.23.2
Gym	0.26.0
PettingZoo	1.20.1
Torch	1.12.1
SortedContainers	2.4.0
Tensorboard	2.11.0
Wandb	0.14.0

Table 5.1: The Python packages used in our experiments.

5.2 Dataset Generation

For each problem and problem size, we use a training set consisting of 5000 instances and a disjoint testing set consisting of 100 instances. These are the very same problem sets used by Kallestad et al. (2023) in their contribution, which they *generated* as a part of their work. Below follows a brief description on generation process.

5.2.1 CVRP

The problem instances was generated according to the methodology presented in Kool et al. (2019), Nazari et al. (2018), but a larger problem variation was also generated. Problem instances containing N orders are generated and grouped together, where $N \in \{20, 50, 100, 200\}$. Both the depot and the order locations are sampled uniformly within the unit square. Furthermore, each order has a associated weight which is set to $\hat{\gamma} = \gamma / D_N$, where γ is randomly sampled from the integers $\{1, ..., 9\}$, and D_N is the normalization factor set to $D_{20} = 30$, $D_{50} = 40$, $D_{100} = 50$, $D_{200} = 50$ for problems with N instances respectively.

5.2.2 PJSP

Problem instances consisting of N jobs are generated and grouped together, where $N \in \{20, 50, 100, 300\}$. Each problem of size N has $\lfloor N/4 \rfloor$ machines at its disposal. Now, each job J_i has a time required to finish it associated with each machine M_j, namely $T_{i,j}$, and it is derived in the following way. The speed of each machine M_j, measured in processing steps per time unit and denoted S_j, is sampled from $\mathcal{N}(\mu, \sigma^2)$ with $\mu = 10$ and $\sigma = 30$, rounded to the nearest integer and set to be at least 1. Each job J_i requires P_i processing steps to complete, a number which is randomly sampled from the set of integers $\{100, 101, ..., 1000\}$. Thus, the time required to finish J_i on machine M_j is then derived by $T_{i,j} = \lceil P_i / S_j \rceil$.

5.3 Baseline Solution Methods

As baselines, we use two different heuristic selection schemes, namely ALNS and DRLH. Within the ALNS framework, we apply two different move acceptance strategies, making it in total three solution methods to act as baseline comparisons to our own contributions. All of these have identical experimental conditions as our own: the initial solutions, the stopping condition and the set of heuristics are all the same. Now we present the details of the baselines.

5.3.1 Adaptive Large Neighborhood Search (ALNS)

With ALNS being one of the most prominent hyperheuristics in literature, it serves as a solid experimental starting point. In particular, we now present two different baseline move acceptance strategies which both are employed on top of the adaptive layer of ALNS.

Simulated Annealing Move Acceptance

Simulated Annealing remains a viable metaheuristic to this day, and its acceptance criteria is used in the original version of ALNS (Ropke and Pisinger, 2006). Thus, the SA acceptance criteria used together with ALNS serves as the main competitive baseline to DRLMA, as well as the general baseline in all of our experiments. We will refer to the ALNS-SA solution method as simply *ALNS*.

Record to Record Travel Move Acceptance

Presented briefly in 2.6.2, the Record to Record Travel move acceptance was shown to outperform the SA acceptance criteria by both Santini et al. (2018) and Hemmati and Hvattum (2017). We thus include the RRT move acceptance as a baseline to further illustrate the effectiveness of DRLMA compared to the No-RL state-of-the-art move acceptances. We will refer to the ALNS-RRT solution method as *RRT*.

5.3.2 Deep Reinforcement Learning Hyperheuristic (DRLH)

As mentioned previously, the DRLH presented by Kallestad et al. (2023) was a huge inspiration to the contribution presented in this thesis. We wish to use their work as a baseline to compare the influence of Deep RL-based heuristic selection compared to Deep RL-based move acceptance, to answer the question of which component plays the biggest part for achieving improvement. Furthermore, DRLH will act as the main competitive baseline in comparison to DRLH+DRLMA.

5.4 Hyperparameter Selection

5.4.1 Adaptive Large Neighborhood Search (ALNS)

For all experiments using the ALNS selection framework, we have set the *reaction factor* to be 0.3 and *segment size* to 50. Now optimally, these hyperparameters should be tuned to the both the specific problem, the size of the problem and the move acceptance employed. To limit the computational scope and save time, these chosen hyperparameters mirrors those used by Kallestad et al. (2023) in their experimentation.

5.4.2 DRLMA

Below, we display the hyperparameters we used when training both DRLMA and DRLH+DRLMA. Table 5.2 displays the general hyperparameters used in both approaches, and for the keen reader we included the hyperparameters of DRLH as a comparison view. Table 5.3 displays the choice of hyperparameters for the reward function when training the DRLMA agent.

Hyperparameter	**DRLH**	**DRLMA**	**DRLH+DRLMA**
#Episodes (max)	5000	5000	5000
SA warmup phase	Yes	No	No
#Epochs	10	10	10
Learning rate	1e-5	1e-5	1e-5
Batch size	64	64	64
Hidden layer sizes	[256, 256]	[256,256]	[256, 256]
Discount rate γ	0.5	0.99	0.99
GAE λ	0.95	0.95	0.95
Clip parameter ϵ	0.2	0.2	0.2
Entropy coefficient	0.0	0.01	0.01
Weight decay	0.0	1e-4	1e-4
KL divergence limit	N/A	0.025	0.025
Value normalization smoothing factor	N/A	1e-5	1e-5

Table 5.2: The general hyperparameters used in our experiments.

Regarding the choice of these particular hyperparameters presented in Table 5.2, the hyperparameter tuning performed was very limited in scope due to the run time of a single

hyperparameter configuration. It seemed reasonable to us to start our experimentation with hyperparameters mirroring those of DRLH. Besides, many of these are related to PPO and take on the default values presented in Schulman et al. (2017), such as the batch size, discount rate γ, GAE λ and the number of epochs used.

Our hyperparameter selection differ slightly from that of DRLH. Firstly, we set the discount rate to 0.99 as this is standard to PPO and seemed to work just fine. Secondly, we included some more schemes into the training that seemed to stabilize learning and improve the final performance of the agent. In the beginning of our experimentation, the agent seemed to very quickly converge to some sub-optimal policy and continued to exploit this policy. To combat this we included an entropy term into the PPO loss as this ensures sufficient exploration (Schulman et al., 2017). We found that L_2-regularization, also known as weight decay, contributed to increased performance, as was suggested by Liu et al. (2020). Another issue we had was the RL phenomenon known as *catastrophic forgetting*, which is when a policy update results in sudden drastic behaviour change, essentially "forgetting" beneficial elements of policy learnt so far and resulting in bad performance. Anecdotal evidence suggests including maximum KL-divergence limit into the training process, essentially skipping any policy update which would make it differ too much from the old policy - their difference measured by the KL-divergence estimation. When including this limit into training, we didn't observe this phenomenon anymore. Lastly, Yu et al. (2022) suggests keeping a running average of the value targets to stabilize the training of the value network. Although their suggestion is intended for the MARL setting, we found this to generally increase (and never hurt) the performance of the trained agent in our single-agent setting.

Hyperparameter	**DRLMA**	**DRLH+DRLMA**
λ (steps ahead)	7	7
ω_{best}	5	3
$\omega_{\text{improvement}}$	0	1
$\omega_{\text{otherwise}}$	-1	0
$\omega_{\text{no improvement \& accept}}$	0	-3
$\omega_{\text{n rejections}}$	0	0

Table 5.3: The hyperparameters of the DRLMA Reward function (4.3).

As mentioned we experienced the choice of reward function to be essential in obtaining a well-performing policy. Table 5.3 shows the selection of hyperparameters for the DRLMA reward function, $R_t^{\lambda \text{ steps ahead}}$, presented in Eq.4.3. After some initial experimentation with different value combinations, these were the configurations that seem to

perform the most optimally across all problems. We attempted to find a single hyper-parameter configuration that would work well in both approaches, but were ultimately unsuccessful. As of the number of steps to look ahead, λ, we briefly experimented with the values $\lambda \in \{5, 7, 10, 15\}$ on CVRP-50, and found that 7 seems to result in the most successful policy.

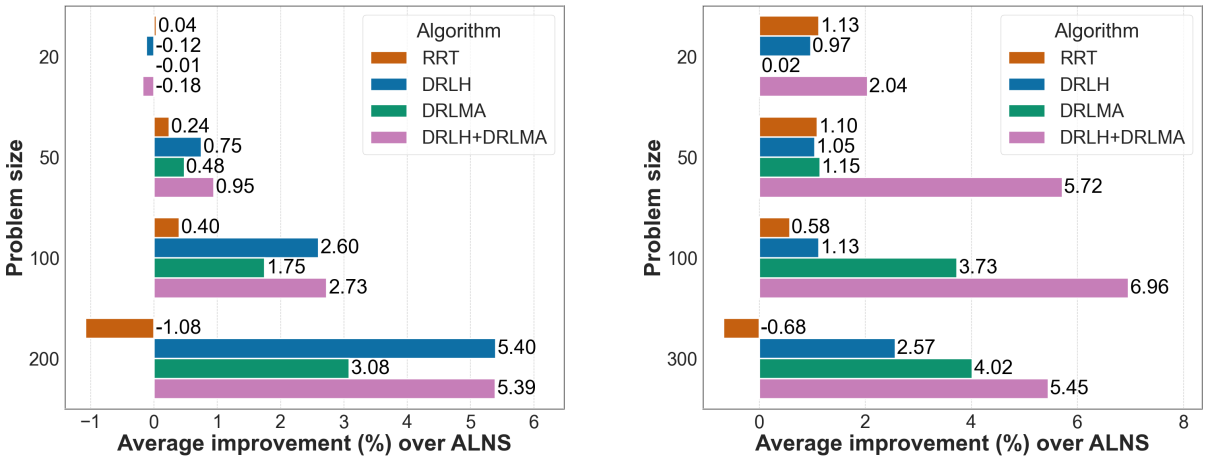
Chapter 6

Results

The performance metrics in the tables and plots in this section is the average (or the median in case of the box plots) of the best solution costs obtained on the 500 test instances of the problem - the same 100 test instances solved 5 times with 5 different input seeds to the random number generator. This strengthens the statistical foundation of our findings, minimizing the particular seed's effect in the calculated performance metrics.

Regarding the tables displaying all performance results, the *Average Cost* column shows the total average performance across the different seeds. The *Best Cost* column represent the *best* average performance among the 5 different seeds, and is included to display the robustness of each solution method. The *Best Cost* shouldn't deviate too far from the *Average Cost*, as the solution method shouldn't be too sensitive to the chosen seed. The last column, named *% Improvement*, shows the percentage of average improvement compared to the performance of ALNS. This is identical to what is showed in Figure 6.1.

Note also that we do not report the running time of the solution methods. Our focus is the effectiveness of an RL-based move acceptance in terms of *solution quality*, and thus we don't find the running time to be of any relevance. Besides, the choice of acceptance criteria has little effect on the overall running time of the solution method, even when backed by a DNN. The curious reader should know that the main bottleneck of our framework in terms of running time is, by far, the low-level heuristics.



(a) CVRP (b) PJSP

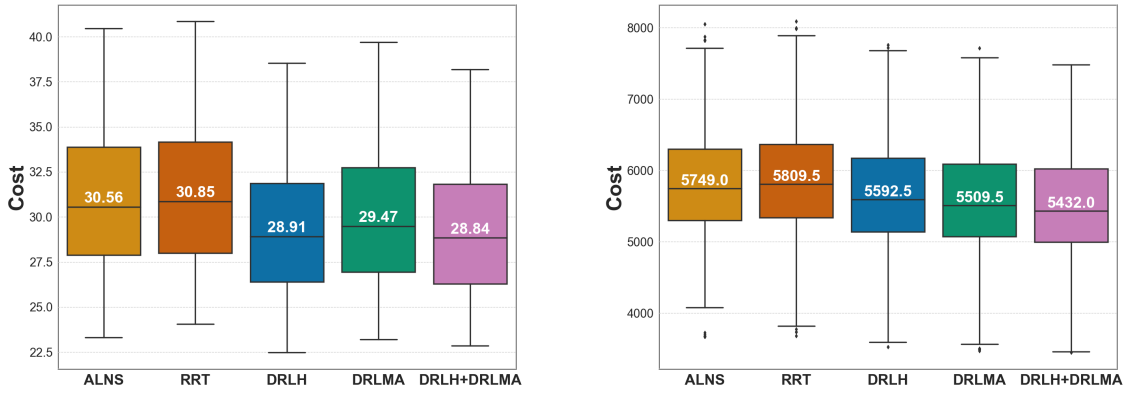
Figure 6.1: Results of DRLMA and DRLH+DRLMA for CVRP and PJSP.

6.1 Results of CVRP

Figure 6.1a shows the improvement in percentage DRLMA, DRLH+DRLMA along with the other baselines has over ALNS on all the sizes of the CVRP. As for DRLMA, it is able to outperform both ALNS and RRT on all sizes except the smallest instances of the problem. However, it is quite clear from the figure that DRLMA is outperformed by DRLH. Also, DRLH+DRLMA is able to keep up DRLH on all sizes except the smallest one, even slightly outperforming it on the size 50 and 100. This suggests that combining DRLMA with DRLH does not hurt the performance, with a slight potential of improving it. Like DRLH, both DRLMA and DRLH+DRLMA follows the clear trend of performing progressively better than ALNS as the size of the problem increases. Figure 6.2a shows the best cost distribution for each solution method on the instances of size 200. This shows a similar trend, with DRLMA outperforming the non-RL baselines, and with DRLH and DRLH+DRLMA further outperforming DRLMA. Overall there is a strong indication that RL-based heuristic selection has a more positive performance effect on CVRP compared to RL-based move acceptance. See Tables 6.1, 6.2, 6.3 and 6.4 for all the results on CVRP of sizes 20, 50, 100 and 200 respectively.

6.2 Results of PJSP

Figure 6.1b shows that both DRLMA and DRLH+DRLMA is able to outperform DRLH on PJSP, with the exception of DRLMA on the smallest problem instances. For this prob-



(a) CVRP-200

(b) PJSP-300

Figure 6.2: Box plot results for the largest instances of CVRP and PJSP.

Solution method	Average Cost	Best Cost	% Improvement
ALNS	6.182	6.179	0.00
RRT	**6.180**	**6.178**	0.04
DRLH	6.190	6.186	-0.12
DRLMA	6.183	6.181	-0.01
DRLH+DRLMA	6.193	6.186	-0.18

Table 6.1: Average performance for CVRP-20.

lem there is strong indication that the solution method benefits more from a sophisticated move acceptance then from a equivalently sophisticated heuristic selection strategy. The degree of improvement of DRLMA clearly increases with the problem size, whereas for DRLH this trend is much more modest. DRLH+DRLMA significantly outperforms the other solution methods, albeit less prominently for the smallest instances. Figure 6.2b illustrates the performance of each solution problem on the largest instances of PJSP, further displaying the effectiveness of the RL-based acceptance criteria on this problem. See Tables 6.5, 6.6, 6.7 and 6.8 for a full overview of performance on PJSP of sizes 20, 50, 100 and 300 respectively.

Solution method	Average Cost	Best Cost	% Improvement
ALNS	10.52	10.51	0.00
RRT	10.50	10.49	0.24
DRLH	10.44	10.44	0.75
DRLMA	10.47	10.46	0.48
DRLH+DRLMA	**10.42**	**10.42**	0.95

Table 6.2: Average performance for CVRP-50.

Solution method	Average Cost	Best Cost	% Improvement
ALNS	16.19	16.18	0.00
RRT	16.13	16.05	0.4
DRLH	15.77	15.75	2.60
DRLMA	15.91	15.87	1.75
DRLH+DRLMA	**15.75**	**15.74**	2.73

Table 6.3: Average performance for CVRP-100.

Solution method	Average Cost	Best Cost	% Improvement
ALNS	30.87	30.77	0.00
RRT	31.20	31.15	-1.08
DRLH	**29.20**	**29.17**	5.40
DRLMA	29.92	29.83	3.09
DRLH+DRLMA	29.20	29.19	5.39

Table 6.4: Average performance for CVRP-200.

Solution method	Average Cost	Best Cost	% Improvement
ALNS	358.4	358.1	0.00
RRT	354.4	354.0	1.13
DRLH	354.9	354.7	0.98
DRLMA	358.3	357.8	0.02
DRLH+DRLMA	**351.1**	**351.0**	2.04

Table 6.5: Average performance for PJSP-20.

Solution method	Average Cost	Best Cost	% Improvement
ALNS	1028.5	1027.5	0.00
RRT	1017.2	1015.5	1.10
DRLH	1017.8	1015.2	1.05
DRLMA	1016.7	1015.8	1.15
DRLH+DRLMA	**969.7**	**969.0**	5.72

Table 6.6: Average performance for PJSP-50.

Solution method	Average Cost	Best Cost	% Improvement
ALNS	1983.5	1981.6	0.00
RRT	1972.0	1970.5	0.58
DRLH	1961.1	1959.6	1.13
DRLMA	1909.5	1900.4	3.73
DRLH+DRLMA	**1845.5**	**1844.5**	6.96

Table 6.7: Average performance for PJSP-100.

Solution method	Average Cost	Best Cost	% Improvement
ALNS	5796.8	5783.6	0.00
RRT	5836.0	5831.0	-0.68
DRLH	5648.1	5646.0	2.57
DRLMA	5563.8	5551.4	4.02
DRLH+DRLMA	**5480.6**	**5477.4**	5.46

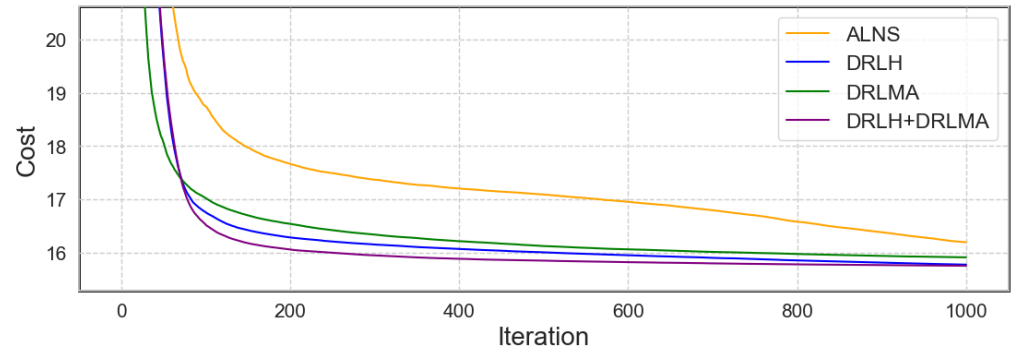
Table 6.8: Average performance for PJSP-300.

6.3 Performance Results

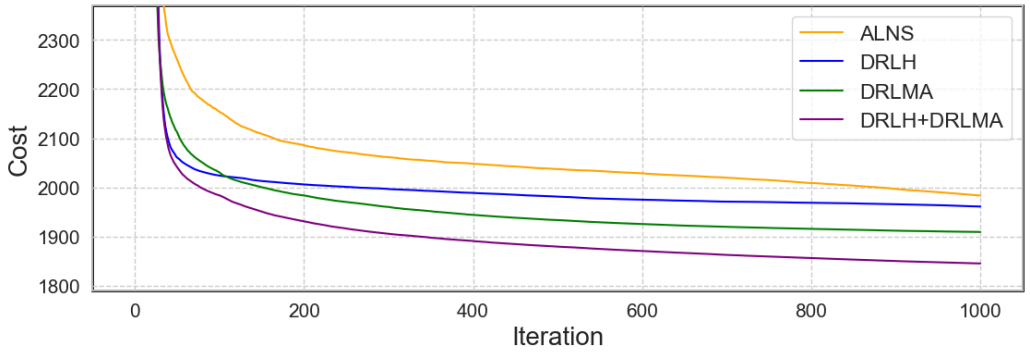
This section takes a closer look on the characteristics of the performance of ALNS, DRLH, DRLMA and DRLH+DRLMA on the two problems. More specifically, we study how the best cost found develops as the search progresses, that is, how quickly a solution method is able to achieve high performance as well as the general performance itself. Figure 6.3 displays the best cost development averaged across all test instances and random number generator seeds. It is apparent from the figure that the solution methods possessing DRLMA is able to discover good solutions early in the search, significantly more quickly than their SA acceptance criteria counterparts. Note that DRLH+DRLMA reaches ALNS-level performance in less than 200 iterations. This suggests that the DRLMA-criteria emphasizes intensification early on in the search to a larger extent than ALNS. However, it is clear from the figure that the strategy is effective, since the DRLMA solution methods is able to reach a notably lower minimum than ALNS by the end of the search. Here we chose size 100 of both problems to illustrate our points, and the performance follows more or less the same pattern for all sizes. We refer the reader to Appendix A for similar performance plots for all sizes of the two problems.

6.4 Move Acceptance Behaviour

Now we wish to gain some insight into the behaviour of the DRLMA move acceptance strategy itself. Figure 6.4 shows how the acceptance probability of ALNS, DRLMA and DRLH+DRLMA changes during the course of the search on CVRP-50. Here, we have only included the probability for uphill moves when the probability is determined by the core part of the acceptance criteria - the Boltzmann function in case of ALNS, and the RL agent in case of DRLMA and DRLH+DRLMA. Figure 6.4a shows the probabilities averaged over all test instances and random number generator seeds. As expected, the



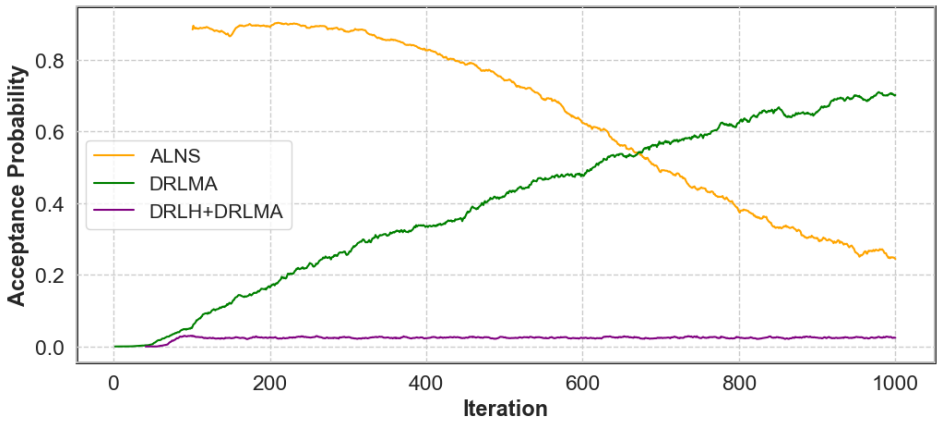
(a) CVRP, 100 orders



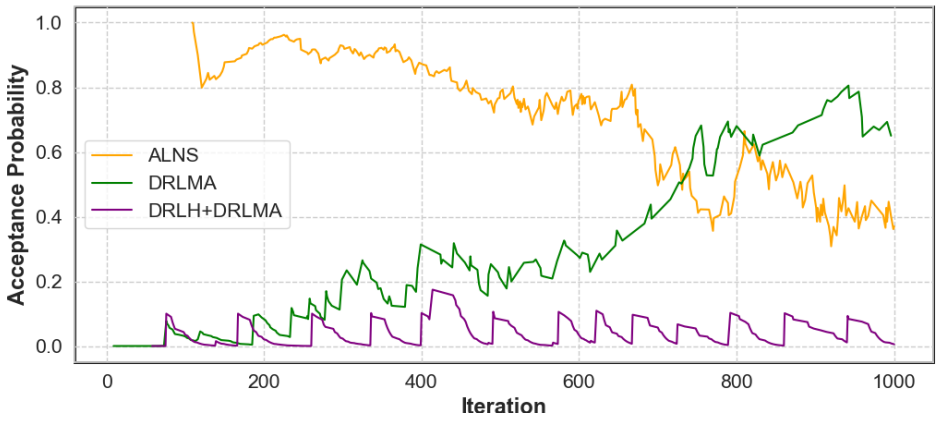
(b) PJSP, 100 jobs

Figure 6.3: Average performance of ALNS, DRLH, DRLMA and DRLH+DRLMA on the two problems.

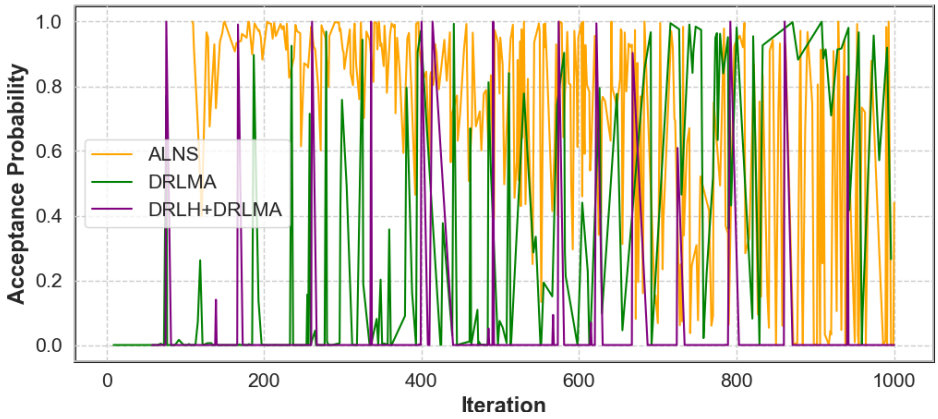
probability of ALNS is slowly decreasing with time - the characteristic of the SA acceptance criteria. Interestingly enough, the behaviour of all three approaches differ quite significantly from each other. For both DRLMA and DRLH+DRLMA, the search starts out with quite low acceptance probabilities, explaining why they are able to find good solutions early on, as observed in Section 6.3. The difference in behaviour of DRLMA and DRLH+DRLMA is likely due to their difference in reward function, as shown in Table 5.3. The DRLH+DRLMA agent is punished quite severely for accepting uphill moves that doesn't lead to future improvement, so the learnt policy becomes very reserved - "sceptical" of candidate solutions. Figures 6.4b and 6.4c shows how the acceptance probabilities changes during the search on some arbitrary instance of the CVRP-50, the former showing a moving average of the probabilities while the latter showing actual probabilities. Although the latter figure seems messy at first glance, it seems clear that learnt policies have low entropy associated with them. Essentially, the agents are very certain in their choices, yielding an acceptance probability close to either 1.0 or 0.0 depending on the situation. The DRLH+DRLMA agent, for instance, gives a probability of close to zero in most cases, and an occasional probability close to 1.0. This seems to indicate that there are certain patterns in the state space that "convinces" the agent that a certain action, either "accept" or "reject", is likely to lead to an improvement in the near future. We chose CVRP-50 to illustrate our points regarding the move acceptance behaviour, but since each problem and problem size corresponds to a unique environment there are no guarantee that the agents learns a similar policy in different settings. However, in our experience the agent shows similar behaviour as this in all of our experiments, although the acceptance probability seems to decrease in general as the size of the problem increases.



(a) Average acceptance probability, smoothed.



(b) Acceptance probability on arbitrary problem instance, smoothed.



(c) Actual acceptance probability on the same problem instance (unsmoothed).

Figure 6.4: Acceptance probabilities for ALNS, DRLMA and DRLH+DRLMA on CVRP-50.

Chapter 7

Conclusion and Future Work

In this thesis we propose DRLMA, a move acceptance for solving combinatorial optimization problems. Located within the hyperheuristic setting, DRLMA consists of a Deep Reinforcement Learning agent trained to decide whether to accept or reject worse candidate solutions produced by the selected low-level heuristic. We claim DRLMA to be a general solution framework in two different senses. Firstly, it can be inserted into any hyperheuristic framework, which we display by combining it with both ALNS and DRLH, producing solution methods we have called DRLMA and DRLH+DRLMA respectively. Secondly, it can be used to solve many different combinatorial optimization problems, as DRLMA bases its decisions on problem-independent information regarding the search process. In our experiments, we strengthen this last claim of generality by solving two different combinatorial optimization problems, namely CVRP and PJSP. We compare the performance of both DRLMA and DRLH+DRLMA with three different baselines, namely ALNS, RRT and DRLH. Our results show that DRLMA is an effective move acceptance, as the two DRLMA-based solution methods is able to outperform their "main competitive" baselines (meaning equivalent hyperheuristic used) in most of the cases, performing about on par with their baselines in the worst cases. Moreover, the increase in performance compared to ALNS seems to be most prominent for large problem instances in general, which indicates that DRLMA is best fit for real-world problem instances. The results also show a general trend of both DRLMA and DRLH+DRLMA finding high-quality solutions earlier on in the search than their baselines. As for comparing Deep RL-based heuristic selection (DRLH) to Deep RL-based move acceptance, our results are mixed, with DRLMA outperforming DRLH on the PJSP and vice versa on the CVRP.

Future research should first and foremost experiment with the effectiveness of DRLMA and DRLH+DRLMA on other combinatorial optimization problems to further strengthen our claim of its superiority and generalization. In spirit of generalization, there is also the incentive to find stable reward design for DRLMA that generalizes to all hyperheuristics, which we simply were unable to achieve in this thesis. Furthermore, we are curious to whether DRLMA would be able to outperform the baselines in the scenario where the stopping criteria for all solution methods are extended to for instance 5000 or 10000 solve iterations. Another potential research direction would be to explore the characteristics of the learnt move acceptance policy beyond our own analysis. In this thesis we have emphasized the importance of balancing intensification and diversification, with the common belief being that diversification is suitable in the beginning of the search and diversification towards the end of the search. However, our learnt policies doesn't seem to behave in accordance with this belief, which directs our curiosity towards the nature of the learnt policy. Now, understanding the decision-making of a Deep RL agent to gain insights into its policy is a challenging task. A natural place to start would be to examine the state-action pairs for clear patters. One could also employ common feature importance techniques, such as studying SHAP values, to identify which features have the most significant influence on the chosen actions. Lastly, we would like to suggest the Multi-Agent RL approach that we have briefly mentioned, meaning training both the heuristic selector agent and the move acceptance agent simultaneously. Its worth mentioning here that we made a brief, naive attempt on this approach using the same hyperparameters that had worked well for both agents in the single-agent setting - but without any luck. Admittedly a weak attempt, we still believe this direction to be a quite challenging one, likely requiring a lot of experimentation and knowledge on topic of MARL. However, we simultaneously believe that the MARL approach has the biggest potential for obtaining a superior solution method, and is thus arguably worth exploring in greater depths.

List of Acronyms and Abbreviations

ALNS Adaptive Large Neighborhood Search.

DDQN Double Deep Q-Network.

DNN Deep neural networks.

DQN Deep Q-Network.

DRL Deep Reinforcement Learning.

DRLH Deep Reinforcement Learning Hyperheuristic.

ILTA Iteration Limited Threshold Acceptance.

LNS Large Neighborhood Search.

MARL Multi-Agent Reinforcement Learning.

MLP Multi-Layered Perceptron.

PPO Proximal Policy Optimization.

RL Reinforcement Learning.

RRT Record-to-Record Travel.

SA Simulated Annealing.

TRPO Trust Region Policy Optimization.

Bibliography

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety, 2016.

R. Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

Mauro Birattari, Luis Paquete, and Thomas Stützle. Classification of metaheuristics and design of experiments for the analysis of components. 03 2003.

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

Peter Cowling, Graham Kendall, and Eric Soubeiga. A hyperheuristic approach to scheduling a sales summit. In Edmund Burke and Wilhelm 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.

Zvi Drezner, Peter M. Hahn, and Éeric D. Taillard. Recent advances for the quadratic assignment problem with special emphasis on instances that are difficult for meta-heuristic methods. *Annals of Operations Research*, 139(1):65–94, Oct 2005. ISSN 1572-9338. doi: 10.1007/s10479-005-3444-z.
URL: https://doi.org/10.1007/s10479-005-3444-z.

Gunter Dueck. New optimization heuristics: The great deluge algorithm and the record-to-record travel. *Journal of Computational Physics*, 104(1):86–92, 1993. ISSN 0021-9991. doi: https://doi.org/10.1006/jcph.1993.1010.
URL: https://www.sciencedirect.com/science/article/pii/S0021999183710107.

Ahmad Hemmati and Lars Magnus 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.

Karla Hoffman and Ted Ralphs. Integer and combinatorial optimization. *Encyclopedia of Operations Research and Management Science*, 01 2013. doi: 10.1007/978-1-4419-1153-7_129.

Holger H. Hoos and Thomas Stützle. Stochastic local search: Foundations and applications. In Holger H. Hoos and Thomas Stützle, editors, *Stochastic Local Search*, The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, San Francisco, 2005. ISBN 978-1-55860-872-6. doi: https://doi.org/10.1016/B978-155860872-6/50018-4.
URL: https://www.sciencedirect.com/science/article/pii/B9781558608726500184.

M Hyde, G Ochoa, T Curtois, and JA Vazquez-Rodriguez. A hyflex module for the maximum satisfiability (max-sat) problem. *School of Computer Science, University of Nottingham, Tech. Rep*, 2010.

Warren G. Jackson, Ender Özcan, and Robert I. John. Move acceptance in local search metaheuristics for cross-domain search. *Expert Systems with Applications*, 109:131–151, 2018. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2018.05.006.
URL: https://www.sciencedirect.com/science/article/pii/S0957417418302835.

Jakob Kallestad, Ramin Hasibi, Ahmad Hemmati, and Kenneth Sörensen. A general deep reinforcement learning hyperheuristic framework for solving combinatorial optimization problems. *European Journal of Operational Research*, 309(1):446–468, 2023. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2023.01.017.
URL: https://www.sciencedirect.com/science/article/pii/S037722172300036X.

Todd R. Kelley. Optimization, an important stage of engineering design. *The Technology Teacher*, 69(5):18–23, 2010.

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

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

Zhuang Liu, Xuanlin Li, Bingyi Kang, and Trevor Darrell. Regularization matters in policy optimization, 2020.
URL: https://openreview.net/forum?id=B1lqDertwr.

Hao Lu, Xingwen Zhang, and Shuang 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.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015. ISSN 1476-4687. doi: 10.1038/nature14236.
URL: https://doi.org/10.1038/nature14236.

David R. Morrison, Sheldon H. Jacobson, Jason J. Sauppe, and Edward C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102, 2016. ISSN 1572-5286. doi: https://doi.org/10.1016/j.disopt.2016.01.005.
URL: https://www.sciencedirect.com/science/article/pii/S1572528616000062.

Mohammadreza Nazari, Afshin Oroojlooy, Lawrence V. Snyder, and Martin Takáč. Reinforcement learning for solving the vehicle routing problem, 2018.

Ender Özcan, Mustafa Misir, Gabriela Ochoa, and Edmund K. Burke. A reinforcement learning-great-deluge hyper-heuristic for examination timetabling. 1(1):39–59, jan 2010. ISSN 1947-8283. doi: 10.4018/jamc.2010102603.
URL: https://doi.org/10.4018/jamc.2010102603.

Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40:455–472, 11 2006. doi: 10.1287/trsc.1050.0135.

Alberto Santini, Stefan Ropke, and Lars Magnus Hvattum. A comparison of acceptance criteria for the adaptive large neighbourhood search metaheuristic. *Journal of Heuristics*, 24(5):783–815, Oct 2018. ISSN 1572-9397. doi: 10.1007/s10732-018-9377-x.
URL: https://doi.org/10.1007/s10732-018-9377-x.

A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Number v. 1 in Algorithms and Combinatorics. Springer, 2003. ISBN 9783540443896.
URL: https://books.google.no/books?id=mqGeSQ6dJycC.

Gerhard Schrimpf, Johannes Schneider, Hermann Stamm-Wilbrandt, and Gunter Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159(2):139–171, 2000. ISSN 0021-9991. doi: https://doi.org/10.1006/jcph.1999.6413.
URL: https://www.sciencedirect.com/science/article/pii/S0021999199964136.

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

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

Paul Shaw. A new local search algorithm providing high quality solutions to vehicle routing problems. 1997.

David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016. doi: 10.1038/nature16961.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.
URL: http://incompleteideas.net/book/the-book-2nd.html.

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

Renata Turkeš, Kenneth Sörensen, and Lars Magnus 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.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

Hao-nan Wang, Ning Liu, Yi-yun Zhang, Da-wei Feng, Feng Huang, Dong-sheng Li, and Yi-ming Zhang. Deep reinforcement learning: a survey. *Frontiers of Information Technology & Electronic Engineering*, 21(12):1726–1744, Dec 2020. ISSN 2095-9230. doi: 10.1631/FITEE.1900533.
URL: https://doi.org/10.1631/FITEE.1900533.

Tony Wauters, Katja Verbeeck, Patrick De Causmaecker, and Greet Vanden Berghe. Boosting metaheuristic search using reinforcement learning. *Studies in Computational Intelligence*, 434:433–452, 01 2013. doi: 10.1007/978-3-642-30671-6-17.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, may 1992. ISSN 0885-6125. doi: 10.1007/BF00992696.
URL: https://doi.org/10.1007/BF00992696.

D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. doi: 10.1109/4235.585893.

Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative, multi-agent games, 2022.

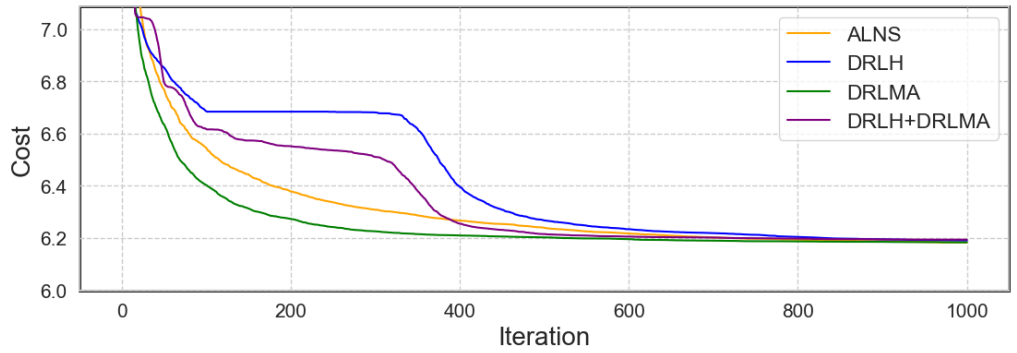
Wei Zhang and Thomas G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, page 1114–1120, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1558603638.

Yuchang Zhang, Ruibin Bai, Rong Qu, Chaofan Tu, and Jiahuan Jin. A deep reinforcement learning based hyper-heuristic for combinatorial optimisation with uncertainties. *European Journal of Operational Research*, 300(2):418–427, 2022. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2021.10.032.
URL: https://www.sciencedirect.com/science/article/pii/S0377221721008821.

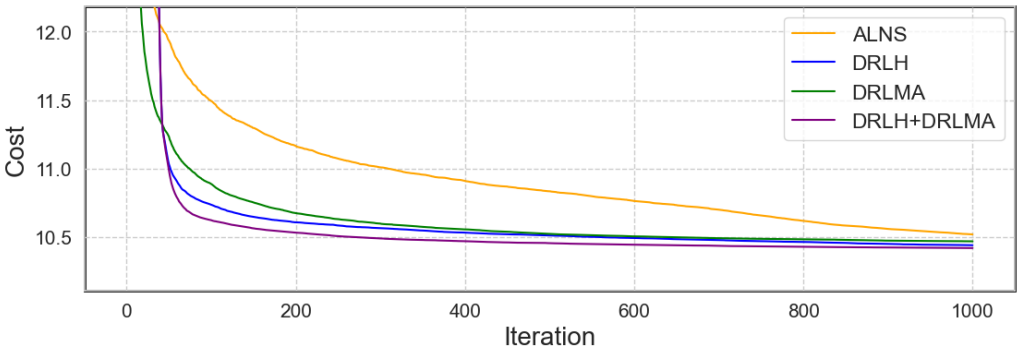
Appendix A

Additional Performance Plots

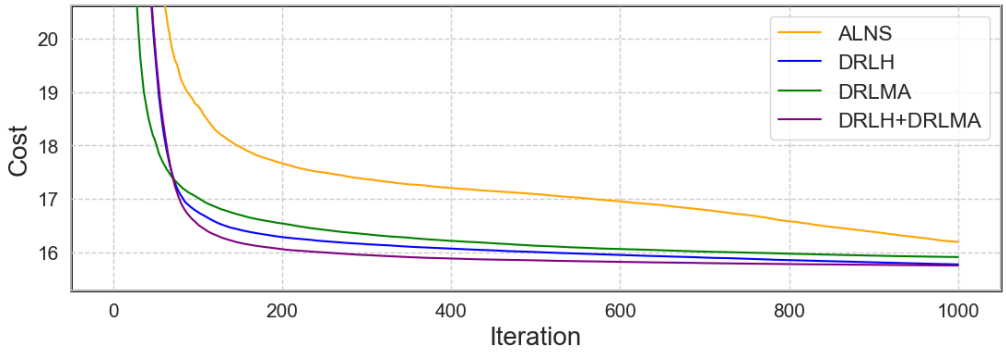
Figures A.1 and A.2 show the performance of ALNS, DRLH, DRLMA and DRLH-DRLMA averaged over the test set for both CVRP and PJSP, respectively. More specifically, they show how the minimum cost develops as the search progresses. Overall, the figures show that using DRLMA as acceptance criterion makes the solution methods, both DRLMA and DRLH-DRLMA, discover good solution faster than their main competitive baselines, namely ALNS and DRLH respectively, as well being able to find better (or as good) solutions overall.



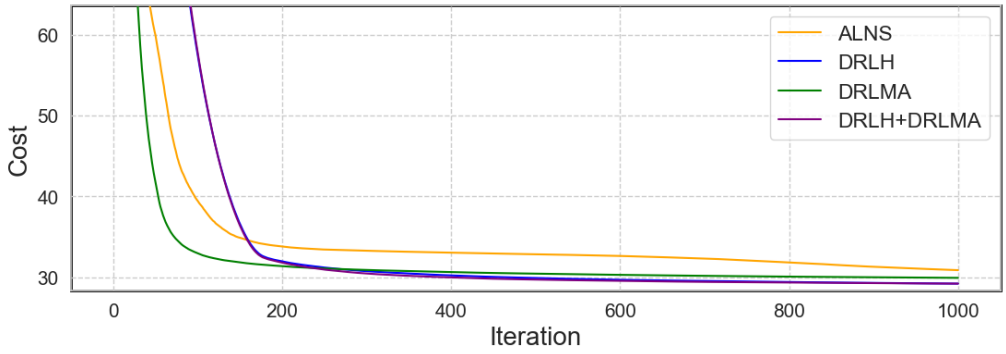
(a) CVRP, 20 orders



(b) CVRP, 50 orders

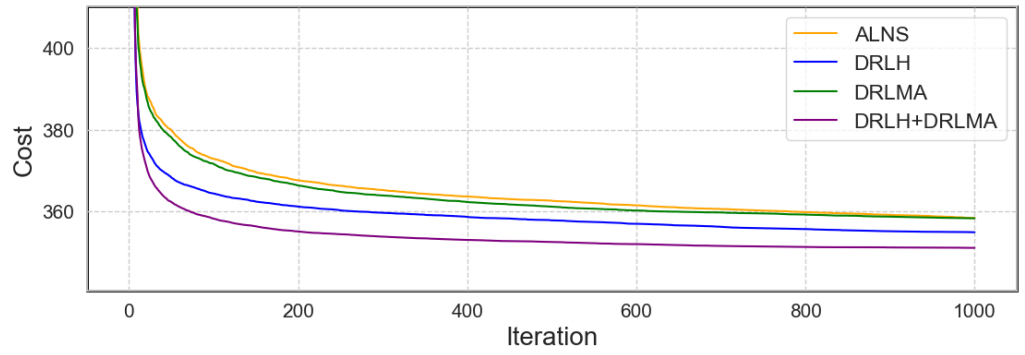


(c) CVRP, 100 orders

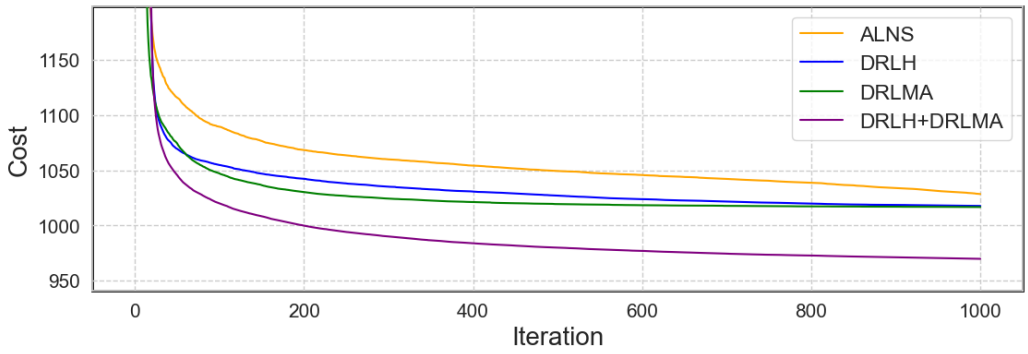


(d) CVRP, 200 orders

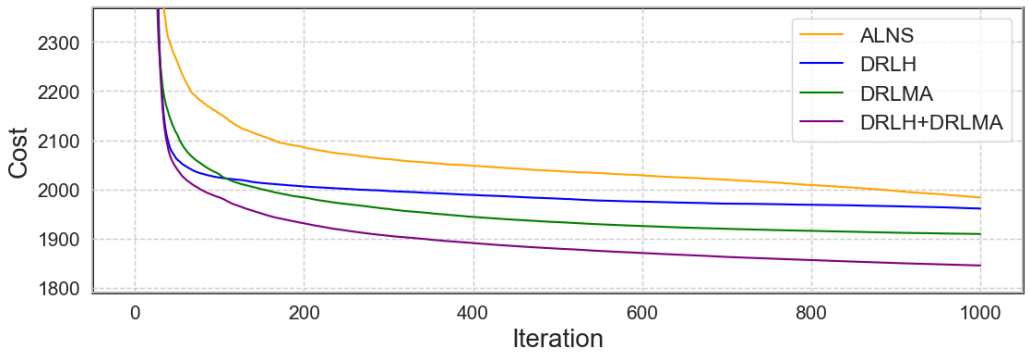
Figure A.1: Average performance of ALNS, DRLH, DRLMA and DRLH-DRLMA on the CVRP.



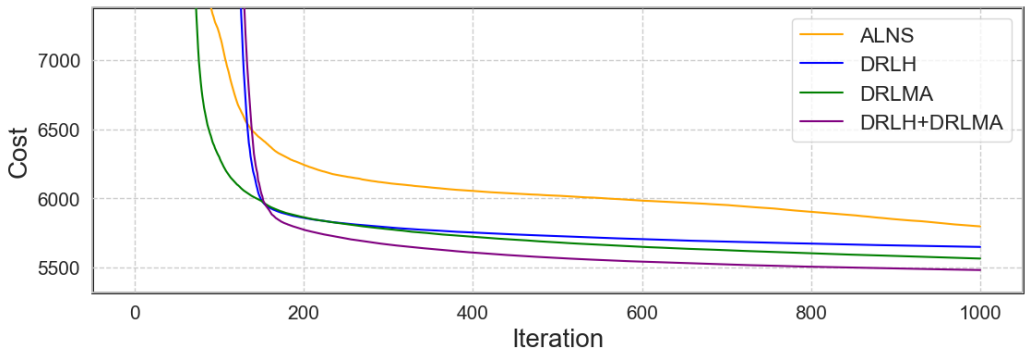
(a) PJSP, 20 jobs



(b) PJSP, 50 jobs



(c) PJSP, 100 jobs



(d) PJSP, 300 jobs

Figure A.2: Average performance of ALNS, DRLH, DRLMA and DRLH-DRLMA on the PJSP.