Junior Management Science

www.jums.academy
ISSN: 2942-1861

Runtime-Sensitive Learned Operator Selection in ALNS: Testing Improvements to Adaptive Operator Selection while Optimizing Runtime

Christopher Dudel

Technical University of Munich

Abstract

We propose and test two variations of the Adaptive Large Neighborhood Search (ALNS) meta-heuristic: First, we add time sensitivity to the operator selection scheme to optimize the ALNS for both solution quality and runtime. We reward comparatively slow operators with reduced rewards for finding improvements. This ensures that the meta-heuristic is slowed down less by operators which consistently find good solutions but take long to do so. Secondly, we replace the Adaptive Layer with a Learned Operator Selection Policy trained via Deep-Q Learning. The training takes both solution quality and operator runtime into account. We test our algorithms against classic ALNS as well as random operator selection. We perform an analysis of how operator portfolios affect performance. Our chosen problem domain is the Capacitated Vehicle Routing Problem with 100 to 400 customer nodes.

Keywords: adaptive large neighborhood search; vehicle routing; optimization; logistics; deep learning

1 Introduction

The Adaptive Large Neighborhood Search (ALNS) is a meta-heuristic proposed by Ropke and Pisinger (2006) which is commonly used to solve Operations Research (OR) problems such as the Capacitated Vehicle Routing Problem (CVRP). Recently, the effectiveness of the main feature of ALNS, the adaptive operator selection, has been called into question (Turkeš et al., 2021). This study identifies potential improvements to ALNS.

ALNS calculates an operator's selection probability based on the value function of explored solutions. Finding, for example, the lowest cost for a routing problem is the end goal of an optimization heuristic. But if finding the optimal solution value were the only goal, one could use exact solvers for any Combinatorial Optimization (CO) problem. The motivation to develop and use heuristic algorithms is to reduce runtimes to manageable levels while still finding acceptable solutions. Following this logic, ALNS can benefit from being able to choose its sub-heuristics based on both their solution qualities and the runtimes it takes to explore them. We sample the available literature already implementing runtime-sensitive ALNS and analyze the impact of operator portfolios on the effectiveness of the runtime-sensitive Adaptive Large Neighborhood Search (tALNS).

Operator runtime is only one additional observation that can be used to better inform the weight update procedure. Yet, building a weight update function that correctly balances the inputs becomes more complicated with each new information. Artificial Neural Networks (ANNs) can be used as function approximators of complex in-out-relationships (Ferrari & Stengel, 2005). (Boualamia et al., 2023) successfully replace the weight update formula with an ANN informed only by the operator reward history within the optimization run.

ALNS goes through a warm-up period, in which the avail-

I would like to thank Dr. Christina Liepold for her exceptional tutor- and mentorship. Her constant support and great advice were immeasurably helpful to this work. I sincerely thank Prof. Dr. Maximilian Schiffer for getting me interested in optimization algorithms and the opportunity to write my thesis, as well as the pragmatic and exciting approach to thesis-writing he promotes. Lastly, I am deeply grateful to Prof. Dr. Josef D. for supporting my studies.

DOI: https://doi.org/10.5282/jums/v11i1pp1-26

able operators are sampled, and operator selection probabilities start to converge towards the actual operator performance (Bongiovanni et al., 2022). Due to the stochastic nature of the operator heuristics, this warm-up can last for a significant portion of the meta-heuristic run. In small problem instances, the run might be over before adequate probabilities have formed. In large problems, operator weights decay towards the lower minimum as finding improvements becomes scarce. These are inherent downsides of the operator selection only being informed by information from previous iteration outcomes in one optimization run (Reijnen et al., 2022). By informing the selection with data from many runs and by training an operator selection policy represented by an ANN with Deep-Q Learning (DQL), we can provide a better base for the operator selection. As this learned policy replaces the adaptive layer, we call this algorithm the Learned Large Neighborhood Search (LLNS).

We differ from previous implementations of tALNS and LLNS by exploring large CVRP problem instances with up to 400 customer nodes. To the best of our knowledge, we provide the most extensive comparison of ALNS and tALNS up to date. We analyze how changing operator portfolios and the inclusion of a Time-Sensitivity Factor affect the performance of the two algorithms. In the field of LLNS, this study is - to the best of our knowledge - the first to optimize operator selection for both solution quality and runtime.

2 State of the Art

In the following, we discuss the literature related to our research: We seek to solve the CVRP - a classic vehicle routing problem domain - utilizing the ALNS meta-heuristic. We pay special attention to approaches that aim to optimize both runtime and solution quality of algorithms. We further discuss how methods from the field of Machine Learning (ML) are used to improve proven OR techniques.

2.1 Capacitated Vehicle Routing Problem

The Capacitated Vehicle Routing Problem (CVRP) is a logistics optimization problem concerned with finding the optimal assignment of a given set of customer nodes to a fleet of vehicles. The vehicles are limited in the number of customers that can be served in one route, for example, via a given carrying capacity. See Godinho et al. (2008) for an in-depth definition and a mathematical formulation of the CVRP. Routing problems have been explored in OR since the 1950s (Dantzig & Ramser, 1959). Vehicle Routing Problems (VRPs) are often expanded with additional details and constraints to make them more applicable to real-world settings, such as heterogeneous fleets (Kucukoglu et al., 2021). Nevertheless, even the basic CVRP is NP-hard and time-consuming to solve via exact methods (Hochba, 1997). Therefore, heuristic approaches are often employed to find acceptable solutions to CVRPs (Gendreau & Potvin, 2010). Heuristic approaches often suffer from getting trapped in local optima. *Meta-heuristics* are a group of algorithms designed to find global optima by combining local improvement heuristics with a higher-level search strategy. They are a powerful method to solve complex CO problems like the CVRP (Gendreau & Potvin, 2010, p. ix).

2.2 Adaptive Large Neighborhood Search

One popular meta-heuristic often applied to routing problems is the Adaptive Large Neighborhood Search (ALNS). (Ropke & Pisinger, 2006) propose ALNS as an extension to the Large Neighborhood Search (LNS) meta-heuristic developed by Shaw (1998). LNS iteratively uses a local search heuristic to destroy and repair an existing solution to find improvements to the objective value. ALNS expands on this approach by allowing sets of destroy and repair heuristics instead of reusing one heuristic in each iteration. A set of operator weights governs which operators are selected for a given iteration. These weights **adapt** in response to the historical performance of the operators within the optimization run. This increases the algorithm's robustness by enabling self-calibration towards using the most efficient operators for a given problem (Pisinger & Ropke, 2007). Section 3.1 and Algorithm 1 give a detailed explanation of the ALNS.

ALNS is successfully used to solve large-scale routing and scheduling problems (Gendreau & Potvin, 2010, pp. 409f.). For an in-depth survey of ALNS applications published until mid-2021, see Windras Mara et al. (2022). They list 228 implementations and find a growing trend for the yearly number of published ALNS studies. This shows that ALNS continues to be seen as an effective optimization framework. (Turkeš et al., 2021) perform a meta-study on ALNS focused on quantifying the average solution quality improvement gained by implementing an adaptive operator selection process. They ask authors of ALNS implementations to re-solve their optimization problems with the same operator portfolios but without weight updates, leading to a random operator selection. They call this version of the meta-heuristic non-adaptive Large Neighborhood Search ((¬)ALNS). (Turkeš et al., 2021) find that, on average, the solution quality of ALNS only improves by 0.14% compared to (¬)ALNS. This relatively low improvement results in the question under which circumstances implementing ALNS over (¬)ALNS is worth the added complexity and whether an improved operator selection mechanism exists (Johnn et al., 2023).

(Turkeš et al., 2021) note two positive outliers in their comparison between ALNS and (¬)ALNS: (Thomas & Schaus, 2018) implement ALNS as a general purpose black box optimizer, set up to be able to solve a wide range of OR problems. For this purpose, they use 30 destroy and 36 repair-heuristics, many more than are commonly pre-selected (Windras Mara et al., 2022). Furthermore, (Thomas & Schaus, 2018) adjust operator weights based on solution quality improvements and operator runtimes. Compared to random operator selection, their ALNS improves solution quality by 15.46%. (Kiefer et al., 2017) manage to improve solution quality by 6.54% over (¬)ALNS. They also utilize more-than-average operators with ten destroy- and eight repair heuristics. Additionally, they also use runtime-sensitivity in their

operator weight update scheme. Both these studies are discussed in more detail in sections 2.2.2 and 2.2.3.

2.2.1 Motivating Runtime-Sensitive Meta-Heuristics

The primary objective of optimization algorithms is to reach a high-quality solution for a given problem. Secondary objectives exist, such as solving within a reasonable time frame, robustness against varying problem characteristics, or low outcome variability (Bengio et al., 2021). Especially in commercial operations, obtaining a good solution fast is often preferred to finding a slightly better solution in a much longer time frame (Ahuja et al., 2002; Wu et al., 2017). Ropke and Pisinger state that their ALNS finds improvements to more than 50% of previous best-known solutions and that it does so "in a reasonable amount of time" (Ropke & Pisinger, 2006, p. 470). The adaptive layer's objective is to choose heuristics that reliably find improvements to the current solution. However, heuristics differ not only in their solution quality but also in their runtime (Ahuja et al., 2002).

(Ahuja et al., 2002) discuss the importance of runtime optimization in large-scale neighborhood searches. Yet few studies are implementing ALNS with an explicit focus on fast solving times. The literature research by Windras Mara et al. (2022) notes the importance of selecting operators that can explore large neighborhoods time-efficiently but does not identify literature that implements time-sensitivity. (Turkeš et al., 2021) focus their meta-analysis of the adaptive layer only on its impact on the solution quality, but note that some authors (analyzed in Table 1) have modified the weight adjustment formula to incorporate operator runtimes (Adulyasak et al., 2014; Canca et al., 2018; Gullhav et al., 2017; Thomas & Schaus, 2018; Wang et al., 2012; Wu et al., 2017). The following section presents a literature review of runtime-sensitive ALNS, focusing on how operator runtime is taken into account in the adaptive layer and on how algorithms can be compared in multi-objective settings where both solution quality and runtime are optimized.

2.2.2 Runtime-Sensitive Operator Selection

Sophisticated heuristics like, for example, *regret_repair* (see Section 3.1.1) - which has to keep track of and repeatedly reevaluate multiple insertion points for each non-inserted node - exceed the runtime of quicker heuristics like *greedy_repair* by a considerable margin. Consider two theoretical heuristics: *fast-repair* and *sophisticated-repair*. *Fast-repair* finds an improvement every tenth iteration. *Sophisticated-repair* finds an improvement in every iteration but takes 100 times longer to complete than *fast-repair*. ALNS would prioritize *sophisticated-repair*, as its higher rewards per iteration would quickly make its operator weight outgrow the weight of *fast-repair*. Given a fixed number of iterations, as is the case in Ropke and Pisinger (2006), this would be the correct approach to reach the best solution quality. However, given a fixed solving time, selecting *fast-repair* more often is preferable.

Table 1 lists the existing research into enhancing ALNS with runtime-sensitive rewards. Note that the lower three papers do not use the ALNS-framework, but the similar Self-Adapting Large Neighborhood Search (SA-LNS). Instead of utilizing destroy- and repair-heuristics, this meta-heuristic iteratively improves a solution via relaxing and re-tightening constraints, mainly in the domain of machine scheduling problems (Laborie & Godard, 2007).

Pairwise or Individual Operator Selection

In classic ALNS, destroy- and repair-operator weights are updated independently of each other (Ropke & Pisinger, 2006). This has the downside of not being able to tell which operator was responsible for the success or failure of the rewarded iteration and whether there are certain destroy- and repair-pairings that work particularly well. This downside is sometimes mitigated by rewarding and selecting destroy-repair-operator-pairs. However, this approach leads to having fewer data points to evaluate since each pair is chosen less often than each independent operator would be (Turkeš et al., 2021).

The underlying problem is further exacerbated in runtime-sensitive ALNS: the chosen destroy-operator can significantly influence the runtime of the repair-operator, especially if the degree of destruction differs between destroy-operators. Within the sampled literature, only Zhang and Yang (2021) evaluate operators in pairs. This is consistent with the works analyzed by Turkeš et al. (2021), out of which 81% use independent operator evaluation. The algorithms proposed in this paper follow this majority. As we utilize an extensive operator portfolio - leading to an excessive number of operator combinations - we select destroy and repair operators individually.

Table 1: Runtime-Sensitive Operator Selection: State of the Art

Reference	Weight-Update Formula for Updated Operator	Explanation
ALNS		
Adulyasak et al. 2014	$(1-\alpha) \times w_{b-1} + \alpha \times \frac{\sum_{i=1}^{\beta} \phi_i}{\Theta \times tf}$ $tf = \frac{\varnothing_b t_{fastest\_operator}}{\varnothing_b t}$	After each batch b of iterations, operator weights w are updated via the gained rewards ϕ. The time factor tf is calculated by dividing the average runtime t within the last batch of the fastest operator with the average runtime of the operator which is currently updated.
Canca et al. 2018	$(1-\alpha) \times w_{b-1} + \alpha \times \frac{\sum_{i=1}^{\beta} \phi_i}{\Theta \times tf}$ tf is unspecified	An unexplained factor tf representing the computational effort required by the operator is used to scale the gained rewards after each batch.
Gullhav et al. 2017	$(1-\alpha) \times w_{b-1} + \alpha \times \frac{\sum_{i=1}^{\beta} rt_i}{\Theta}$ $rt_i = max(\phi_{rej.},\ \phi_i \times min(1, \frac{T_{avg}}{t_i \times \omega}))$	Each reward is scaled by a normalized factor derived by dividing the average operator CPU time by the observed operator CPU time. The observed CPU time is further scaled by a meta-parameter ω (tuned in range 0.5-1.5). The reward can only be lowered for slower-than-average operators. Rewards can also never be lower than the reward gained for finding a rejected solution $\phi_{rej.}$.
Kiefer et al. 2017	$(1-\alpha) \times w_{b-1} + \alpha \times \frac{\sum_{i=1}^{\beta} \phi_i}{\Theta \times tf}$ $tf = \frac{\varnothing t_{reference\_operator}}{\varnothing_b t}$	Runtime-factor tf is calculated by dividing the average runtime of a pre-chosen reference operator by the average runtime of the updated operator within the last batch. tf is only present for repair-operators.
Wu et al. 2017	$(1-\alpha) \times w_{i-1} + \alpha \times \frac{\phi_i}{tf}$ tf is unspecified	The reward gained for finding a best or accepted solution is scaled with an undisclosed time cost tf.
Zhang and Yang 2021	$(1-\alpha) \times w_{i-1} + \alpha \times \frac{\phi_i}{tf_i}$ tf_i is runtime of iteration i	The adaptive layer rewards combinations of destroy- and repair-operators. Rewards are scaled with the execution time of the iteration.
SA-LNS		
Laborie and Godard 2007	$(1-\alpha) \times w_{i-1} + \alpha \times \phi_i$ $\phi_i = \frac{\triangle c_i}{\triangle t_i}$	Probabilities for relaxation and completion strategies are updated each iteration via a ratio derived from the solution improvement (if any) $\triangle c_i$ and the CPU cycle time $\triangle t_i$. How the CPU time relates to the solution changes or whether either of the two inputs is normalized is not explained.
Thomas and Schaus 2018	$(1-\alpha) \times \frac{L}{T} \times T_o + \alpha \times L_o$ $L_o = \frac{\sum_{i^*}^{i} \triangle c_{io}}{\sum_{i^*}^{i} \triangle t_{io}}$ $T_o = \frac{\sum_{0}^{i} \triangle c_{io}}{\sum_{0}^{i} \triangle t_{io}}$ L and T likewise but over all operators	An expansion on SA-LNS updating weights each iteration, but over a rolling time window starting slightly before the iteration of the current best-known solution (i^*). Within this window, the local efficiency of each operator L_o is calculated. The total operator efficiency T_o, which is taken over all iterations, serves to smooth out the weight changes. The total efficiency is further scaled by the ratio between local and total efficiency over all operators (L and T).
Wang et al. 2012	$(1-\alpha) \times w_{i-1} + \alpha \times \frac{\phi_i}{tf_i}$ tf_i is the computational time in the current iteration	No information is given which operations within the iteration count towards tf.

α: weight update factor. β: size of batch. Θ: operator appearances in batch.

Table 2: Algorithm Acceptance Criteria: State of the Art

Reference	Stopping Criterion	Comparison
ALNS		
Adulyasak et al. 2014	100 iter. no new accepted / 300 iter. no new best*	Both solution quality and runtimes are reported independently. Additionally, average operator performances and runtimes are reported.
Canca et al. 2018	3,600 seconds	Algorithms are compared by best solution cost reached within stopping criterion.
Gullhav et al. 2017	maximum iterations	The maximum iterations are set so that an average run takes slightly longer than 15 minutes. Average solution GTOs are compared at 5, 10 and 15 minutes.
Kiefer et al. 2017	480 seconds	Report the average cost over multiple runs, and best cost found for each problem instance. They judge that their proposed algorithm outperforms a competitor because it finds better solutions than the competitor in more instances.
Wu et al. 2017	maximum iterations	Stopping criterion is set dependent on problem size, with larger problems having less iterations to limit solving times. Both time and solution quality are reported and compared qualitatively. The authors justify achieving worse solutions than comparable algorithms because they limited themselves to runtimes of one hour.
Zhang and Yang 2021	$I_{max} = 25,000 + 50\times$ instance size	Solution quality and runtime are independently reported, but not compared.
SA-LNS		
Laborie and Godard 2007	max. consecutive non-improving iter.	Stopping criterion is dependent on problem size. Algorithms are judged solely by final solution quality. Solving times are not reported.
Thomas and Schaus 2018	240 seconds	Present a figure showing average solution costs at solving times for their proposed algorithm and a competitor. Their algorithm outperforms the competitor at all times during the 240 seconds.
Wang et al. 2012	max. consecutive non-improving iter. / 600 seconds*	Report the solution gap to optimum (calculated exactly with CPLEX). If the gap to optimum is zero, a time savings in seconds to the runtime of CPLEX is reported.

I_{max} refers to the maximum number of iterations.

*Whichever is reached first

2.2.3 Algorithm Comparison Criterion

The challenge of multi-objective comparisons between heuristics appears again when comparing the overall performance of meta-heuristics. The fundamental question that needs to be answered is: how much runtime is a given increase in solution quality worth?

Table 2 shows how the authors exploring runtime-sensitive ALNS and SA-LNS compare different meta-heuristic implementations. Comparing solution quality after a fixed runtime is the preferred method in the literature sampled. However, some drawbacks are associated with this approach, which are discussed in Section 4.2. Instead, we opt to inverse the two optimization objectives and compare the elapsed runtime upon reaching a fixed solution quality.

2.3 Combining Machine Learning with Meta-Heuristics

Recently, multiple approaches to utilize Machine Learning (ML) for solving classical CO problems have been proposed (Bengio et al., 2021; Karimi-Mamaghan et al., 2022; Reijnen et al., 2022). The following section starts with an overview of the field of ML in CO and proceeds with a deep-dive into the literature studying a learned operator selection policy for ALNS.

2.3.1 Machine Learning in Combinatorial Optimization

Combinatorial Optimization (CO) problems are often time-intensive to solve, and the field is constantly searching for improved algorithms to optimize hard problems in less time. One improvement step was the development of powerful meta-heuristics. A current research direction is to exploit ML-techniques, which have recently significantly improved in performance and spread of adoption. (Bengio et al., 2021) present an overview into the combination of CO and ML.

A straightforward approach to utilize ML for CO is to directly learn solutions with a given OR problem as input: (Nazari et al., 2018) do so for the VRP, finding solutions for instances with up to 100 nodes. (Kool et al., 2018) train an Attention Model to optimize the Traveling Salesman Problem (TSP) and CVRP. Currently, these direct approaches suffer from exceedingly long runtimes when scaling to larger instances (Reijnen et al., 2022).

A second idea is to combine a proven optimization approach with a specialized learned component. Several authors (Song et al., 2020; Sonnerat et al., 2021; Wu et al., 2021; Zhou et al., 2023) enhance exact solvers of Linear and Mixed Integer Problems with ML, for example by selecting constraints to relax via a learned policy. (Da Costa et al., 2021) take a heuristic approach and pick nodes for 2-opt moves via Deep Reinforcement Learning (DRL).

Meta-heuristics have proven efficient at optimizing large-scale OR problems (Gendreau & Potvin, 2010). Increasingly, ML techniques have been combined with meta-heuristics to create powerful optimization algorithms (Karimi-Mamaghan et al., 2022; Oliva et al., 2021). A review of the current state of the art for meta-heuristics enhanced by ML can be found in Karimi-Mamaghan et al. (2022). For example, (Joe & Lau, 2020) combine DRL with Simulated Annealing (SA) for the Dynamic CVRP. The LNS by Shaw (1998) has been a popular meta-heuristic to utilize around a ML operator: (Gao et al., 2020) chose promising VRP neighborhoods by using a trained agent as the destroy operator. (Wu et al., 2022) learn an LNS improvement heuristic for TSP and CVRP. (Hottung & Tierney, 2022) propose a Neural Large Neighborhood Search with a learned improvement heuristic for CVRP, Split Delivery Vehicle Routing Problem, and Capacitated Team Orienteering Problem.

Partially motivated by the findings of Turkeš et al. (2021), another approach has emerged which seeks to improve the adaptive layer of ALNS with a learned operator selection. The meta-heuristic is modeled as a Marcov Decision Process (MDP), where each step entails selecting the most efficient operators for the current iteration (Johnn et al., 2023). We identify four studies (Bongiovanni et al., 2022; Boualamia et al., 2023; Johnn et al., 2023; Reijnen et al., 2022) adapting this approach and explore them in Section 2.3.2. The field of selecting the best algorithm to solve a given combinatorial optimization problem is called Automated Algorithm Selection (Kerschke et al., 2019). Reviews can be found in Kerschke et al. (2019) and Karimi-Mamaghan et al. (2022). We further identify two papers from this field that do not utilize the ALNS-framework but are close enough in their approach to warrant mentioning: (Chen & Tian, 2018) propose an actor-critic neural network to rewrite feasible solutions of Job Scheduling and VRP problems with up to 100 customers. Two policies are trained: A region-picking policy that chooses which features of the previous solution to remove and a rule-picking policy that determines how the next solution is created. (Lu et al., 2020) propose a novel heuristic framework that iteratively improves a feasible solution. Operators are split into two pools: perturbation and improvement. A meta-controller decides which pool to choose from for the next iteration. An attention network is used to train the selection of the improvement operators, while perturbation operators are chosen randomly. They use this framework to find solutions for CVRP-instances of size 100.

2.3.2 Learned Large Neighborhood Search

The approach by Boualamia et al. (2023) stays comparatively close to classic ALNS: Instead of training a policy over multiple optimization runs, they opt to replace the ALNS weight update and roulette wheel selection scheme with an epsilon-greedy policy learned anew within each optimization run. The Q-Table, which effectively replaces the operator weights, is updated via rewards based on the acceptance check outcome of each iteration, as is the case in ALNS. They achieve exploration of destroy-repair-pairs by starting with a chance of $\varepsilon = 0.6$ to select a random pairing. This chance exponentially decays to exploit pairings that have proven efficient. As their approach requires no training, it can be applied to broad ranges of problem instances, demonstrated by optimizing CVRP instances between 45 and 1200 nodes.

(Bongiovanni et al., 2022) optimize the Autonomous

Ridesharing Problem. They build a dataset of 1.5 million transitions, which they use for offline training of an ensemble of Random Forrest classification predictors. Each pair of destroy- and repair operators is associated with a Random Forrest model, which predicts whether the pair will likely find an improvement to the active solution. A roulette wheel is then used to draw from all pairs which are deemed suitable. The authors utilize a large feature portfolio, including detailed domain-specific observations such as the current average state of charge of all vehicles. The predicted reward is the improvement to the solution quality after using the operators for one iteration.

(Johnn et al., 2023) exploit extensive knowledge about the search state to better inform the operator selection. They train a selection policy via Deep-Q Learning, testing both a Multi-Layer Perceptron and a Graph Neural Network as function approximators. To inform the selection, they translate both the CVRP instance and the current solution state - represented by the existing routes - into graphs, which they pass as features. A single reward is given at the end of each run, calculated as the improvement in solution quality from the initial to the best solution. Training is done on 20-node CVRP instances, with results showing that the trained policies still outperform ALNS in larger problems. The authors find that the learned policy leads to greater improvements over ALNS with growing operator portfolios.

(Reijnen et al., 2022) take a different approach to Johnn et al. (2023) and Bongiovanni et al. (2022) in that they only consider domain-agnostic features, such as the number of iterations since the current best solution was found. They use Deep Reinforcement Learning to train a pairwise operator selection policy. After 32 hours of training on instances of size 20, their agent is able to outperform ALNS on instances of size 100. Their chosen domain for the comparison is the Time-Dependent Orienteering Problem with Time Windows.

Table 3 gives an overview of the relevant features of the sampled studies.

Table 3: Learned Large Neighborhood Search: State of the Art

Reference	Learning Technique	Num. Ω^-/Ω^+	Pairwise	Instance Size	Dimension	Num. Specific / Agnostic Features	Rewards
Bongiovanni et al., 2022	Random Forrest Ensemble	3 / 3	✓	Autonomous Ridesharing	24.000 requests over 7 days	40 / 6	Objective Improvement in Iteration
Boualamia et al., 2023	Q-Learning with Epsilon-Greedy Policy	4 / 2	✓	CVRP	45 - 1200	- / -	ALNS rewards
Johnn et al., 2023	DQL, Multi-Layer Perceptron & Graph Neural Network	12 / 2	X	CVRP	20 - 100	Instance graph & route graphs / -	Solution quality increase from first to last iteration
Reijnen et al., 2022	Deep Reinforcement Learning	4 / 3	✓	Orienteering Problem	20 - 100	- / 8	ALNS rewards
This LLNS	DQL, C51	5* / 7	X	CVRP	100 - 399	- / 10	Multi-Objective: solution quality & runtime

Ω^- / Ω^+ refers to the sets of destroy / repair operators. *Only 1 destroy operator is used in Algorithm Comparison.
Pairwise refers to whether destroy and repair operators are chosen as a pair, or independently.
Specific / Agnostic refers to whether or not a feature is independent from the problem domain.
C51 is a variation of Q-Learning where the expected Q-value is represented by a distribution (Bellemare et al., 2017).

2.4 Contribution

The presented literature review shows that significant efforts have been made to improve the ALNS adaptive layer. Nevertheless, there exists a need for further study: in the area of runtime-sensitive ALNS, we note that none of the authors in our review put the focus of their work onto the time-sensitivity. Section 4.6 will present an in-depth comparison of ALNS and tALNS, including discussing an edge-case where tALNS can perform worse. We prove that a tuned Time-Sensitivity Factor ω, previously only used by Gullhav et al. (2017), is vital to the performance of tALNS. No analysis of this kind exists in the sampled literature.

In the case of LLNS, we differentiate on three levels: First, we expand the instance size range from the commonly used VRP-100 up to 400 nodes, with 700+ node instances used to test generalization. Furthermore, we focus on using domain-agnostic observations. This opens up the possibility to use the LLNS-framework outside of CVRP. Lastly, to the best of our knowledge, this study is the first to combine runtime-sensitivity and learned operator selection. This presents a significant challenge for training, as training results depend on a complex multi-objective reward function, which has to be well-tuned to allow learning a time-efficient operator selection.

3 Methodology

As we evaluate two potential improvements to ALNS, the Methodology section is split into two parts. Section 3.1 explains the process of classic ALNS and our proposal of tALNS. Section 3.2 details how we learn the operator selection policy with ML

3.1 Runtime-Sensitive Adaptive Large Neighborhood Search

To make the results of this study comparable to other uses of ALNS, an effort was made to not deviate from a standard implementation of ALNS wherever feasible. We use the most prominent Adaptive Mechanism, Acceptance Criterion, and Termination Criterion as surveyed by Windras Mara et al. (2022). Their literature review shows that, out of 251 surveyed ALNS implementations, 99% use a Roulette Wheel Adaptive Mechanism, 81% use a Metropolis Acceptance Criterion (such as SA) and 75% use a maximum number of iterations as the Termination Criterion. Figure 1 shows how the solution of a small CVRP-instance might evolve over the course of an ALNS-run.

Algorithm 1 gives an overview of the process of the metaheuristic. All relevant details are explained in depth within this chapter.

Central to Algorithm 1 is a solution S. It is initialized via a sweep operation, sorting nodes into feasible routes based on their angular position. Depending on the problem size and layout, sweep tends to find solutions between 25% and 10% distance to the optimal solution for our instances. Sampling the sweep results shows it performs better in problems that place the depot at the center. In addition to the sweep, we run one iteration each for every available repair heuristic. This is done to be able to calculate a starting temperature sat_0 for SA (Kirkpatrick et al., 1983).

This solution-initialization, which is marked as *sweep+* in Algorithm 1, leads to the creation of three tracked solutions: The active solution S, the best-known solution S^*, and an accepted solution S^a. They allow performing an acceptance check after each iteration i utilizing SA. The probability of accepting a worse solution is governed by the SA temperature sat. Section 3.1.2 explains the SA-setup.

ALNS differentiates from LNS by utilizing sets of destroy- and repair operators Ω^- and Ω^+. In each iteration, a destroy- and a repair operator $o^{-/+}$ are chosen via a roulette wheel selection dependent on operator weights w. After the repair phase, a local optimization is performed on the active solution. Section 3.1.1 explains the operators and the local optimization.

Algorithm 1: Algorithm for the (time-sensitive) Adaptive Large Neighborhood Search

```
Inputs: problem-instance 𝒾, meta-parameters 𝒫
Initialization:
S*, sat ⟵ sweep+_initialize(𝒾)
w ⟵ weight_initialize(𝒫, Ω⁻, Ω⁺)
S^a = S*
Iterations:
for i = 1, ..., I_max do
    S = S^a
    o⁻, o⁺ ⟵ choose(w)
    S, t⁻ ⟵ destroy(S, o⁻)
    S, t⁺ ⟵ repair(S, o⁺)
    S ⟵ local_optimization(S)
    S*, S^a, ρ ⟵ acceptance_check(S, S*, S^a, sat)
    Φ, ct^{-/+}, Θ^{-/+} ⟵ score_update(Φ, ρ, t⁻, t⁺)
    if i % β == 0 then
        w ⟵ weight_update(w, Φ, ct, Θ)
        Φ, ct, Θ = 0 for each o
return S*
```

Based on the result of the acceptance check ρ, the active operators scores Φ are increased by a reward value of ϕ. Furthermore, we keep track of each operator's number of appearances Θ and cumulative runtime ct per batch. The operator weights are updated after a fixed batch b of iterations. The iterations within the next batch will select their operators based on the updated weights. Section 3.1.3 explains the weight update mechanism. Finally, after the stopping criterion is reached - in this case, a maximum number of iterations - the currently best-known solution is presented as the final result.

Based on the result of the acceptance check ρ, the active operators scores Φ are increased by a reward value of ϕ. Furthermore, we keep track of each operator's number of appearances Θ and cumulative runtime ct per batch. The operator weights are updated after a fixed batch b of iterations. The iterations within the next batch will select their operators based on the updated weights. Section 3.1.3 explains the weight update mechanism. Finally, after the stopping criterion is reached - in this case, a maximum number of iterations - the currently best-known solution is presented as the final result.

3.1.1 Operators

This section will detail the operators used within the algorithm implementation. The selection of operators aims to allow both exploitation and exploration of the search neighborhoods and offers a mix of faster and slower operators. A

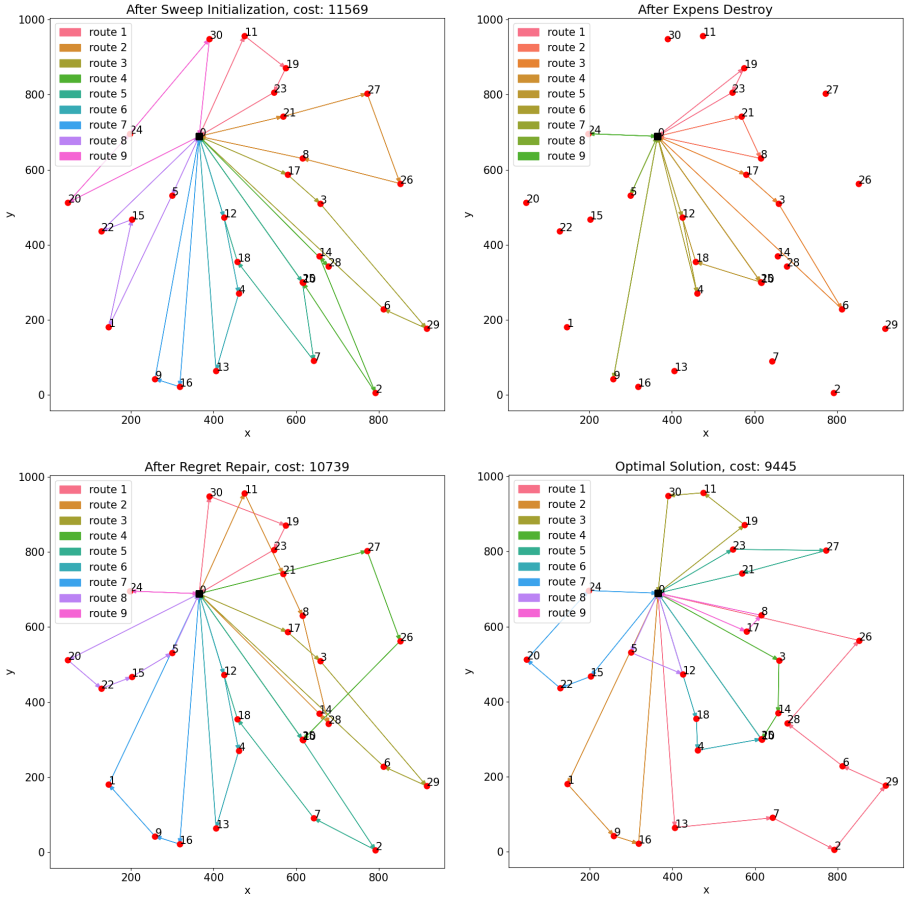

Figure 1: Four Example Solutions during an ALNS Run. Top left: initial solution, top right: destroyed solution, bottom left: repaired solution, bottom right: resulting optimal solution.

list of the most commonly used ALNS operators can be found in Windras Mara et al. (2022, p. 9).

Destroy Operators

The algorithm employs five destroy operators. Their individual degrees of destruction are governed by a meta-parameter detailed in chapter 4.4. Operators remove nodes until the desired degree of destruction is reached. The operators are as follows:

- **random_destroy** randomly removes nodes until the desired degree of destruction is reached.
- **nearby_destroy** randomly removes a single node. Additional nodes with the shortest edge distance to the initial node are removed.
- **demand_destroy** randomly removes a single node. Additional nodes with the lowest difference in demand compared to the initial node are removed.
- **expens_destroy** calculates the cost-savings that could be realized by removing a node. Nodes are removed in order of the highest costs.
- **routes_destroy** picks a random route and removes all nodes within it from the solution. It then tests if the desired degree of destruction is reached. If not, additional routes are removed. This operator can exceed the degree of destruction depending on the size of the last removed route.

Repair Operators

The repair operators are passed a list of removed nodes. The list is in order of removal. The repair operators will try to reinsert the removed nodes into feasible routes until all nodes are back in the solution. If no feasible route can be found, a new route is created. The operators are as follows:

- **greedy_repair** shuffles the list of removed nodes. Then, it finds the cheapest feasible route and position for the first node and inserts it into the solution. This is repeated until all nodes are repaired.
- **nearby_repair** takes the first node from the list of removed nodes. It then searches for the closest node (by edge distance) within a route feasible for insertion. The best position within this route is calculated, and the node is inserted. This is repeated for all removed nodes.
- **locost_repair** calculates the insertion cost and best position in each route for each node. It then places the node with the lowest insertion cost. After each insertion, the costs for inserting into the recently changed

route are recalculated, and all remaining nodes are re-sorted.

- **sorted_repair** runs the same initial sorting procedure as *locost_repair*. However, no recalculations are done. Nodes are inserted in the initial order and only check if their preferred route still has enough capacity. If not, the feasibility test is repeated until a valid route is found or a new route is created. While this operator was intended to be a faster, less precise version of *locost_repair*, testing shows that it often performs slower due to nested validity checks (see Figure 4).
- **regret_repair** calculates the cheapest and second-cheapest feasible insertion route for each removed node in the current solution. The node with the highest regret value, calculated as the difference between the second-cheapest and cheapest route, is inserted first. Insertion costs are recalculated after each placement, taking the recently placed node into account. This is repeated until all nodes are placed. The goal of *regret_repair* is to first insert nodes that would incur significant cost increases if their optimal route was filled up with other nodes.
- **regrdk_repair** spells out as Regret-Random-k Repair. k specifies which route's insertion cost is chosen for calculating the regret cost. For example, if k=4, the regret value will be the forth-cheapest route insertion cost minus the cheapest route insertion cost. Compared to k=2 regret repair, it favors nodes that might have two relatively cheap routes but a steep price increase afterward. This operator randomly chooses k so that $2 \leq k \leq min(5,\ number\ of\ routes)$.
- **friend_repair** is a complex insertion algorithm that calls other repair operators. It is based on finding a close-to-optimal solution in small neighborhoods (which we call friend-groups). First, we calculate the insertion cost for every removed node into every existing route. For each node, the best three routes are stored. A matrix saves a friend-score for each node pair by calculating the overlap in their three cheapest routes. For example, node 1 has cheapest insertions in routes A, B, and C. Node 2 is cheapest in routes B, C, and D. The overlap is 2. The node with the highest combined friend-score with all other nodes is chosen as the first friend. The three nodes with the highest friend-scores with the center-node are added to the friend-group, and those four nodes in the friend-group are removed from the list of removed nodes. The process of creating friend-groups is then repeated two more times.
  Now, we repeatedly call *greedy_repair* on a temporary copy of the active solution, with each permutation of the order of nodes in a friend-group as the nodes-to-insert. Since each friend-group consists of four nodes, 24 permutations are tested for each friend-group. For this use case, *greedy_repair* is set to not shuffle the order of the nodes-to-insert. The solution cost after insertion is tracked for each permutation, and the cheapest permutation is used on the active solution. This is repeated for each friend group. To insert the remaining removed nodes not part of any friend-group, *regret_repair* is called.
  The meta-parameters present in *friend_repair* have a massive influence on the runtime and performance of the operator. The n_r routes used for the overlap friend-score influence how far away from the cheapest route friends are noticed. The number of friend-groups and the number of nodes in each friend-group directly influence the runtime. In the case of the n_g friend-groups, the increase is linear, with the sorting and *greedy_repair* phase having to be repeated. In the case of the n_f friends per group, the number of permutations is multiplied by N whenever an N-th friend is added. For this implementation, the parameters were set as $n_r = 3$, $n_g = max(3, n_{removed\_nodes}//n_f)$, and $n_f = 4$.

After the repair operation, a route-internal local 2-opt operation is run on every route that changed within the iteration. Positions of nodes within the route are swapped until no more improvements can be found. As the repair operators create solutions with varying levels of local optimization, the time needed for the 2-opt is added to the runtime of the repair operator. See Da Costa et al. (2021) for an in-depth explanation of the 2-opt heuristic.

3.1.2 Acceptance Criterion

As discussed in depth in Santini et al. (2018), ALNS uses an acceptance criterion after each iteration to decide which solution to proceed with. The criterion chosen for this implementation is Simulated Annealing, developed by Kirkpatrick et al. (1983) and Černý (1985). A comparison to other acceptance criteria can be found in Santini et al. (2018). Since this component of the meta-heuristic stays the same for adaptive- and learned operator-selection, fine-tuning it was not of high priority. Formula (1) shows the probability p to accept a new solution given a difference $\Delta c = c_S - c_{S^a}$ between the cost of the new solution and the accepted solution.

$$p = \begin{cases} 1 & \text{if } \Delta c \leq 0 \\ e^{\frac{-\Delta c}{sat_i}} & \text{if } \Delta c > 0 \end{cases} \tag{1}$$

Since the meta-heuristic has to be able to solve instances with significantly different cost ranges, the temperature is initialized after running the sweep-initialization as well as one iteration for each repair operator. Following Ben-Ameur (2004) and Kirkpatrick et al. (1983), the difference between the best and worst solution of these initial runs is used as the initial temperature.

The temperature is lowered each iteration by multiplication with a cooling rate $sa\_cr < 1$, which gets initialized via formula (2), dependent on the maximum iterations I_{max}, as well as two meta-parameters governing the target remaining temperature sa_tp (as a percentage of the starting temperature) in a target iteration sa_ti. This approach ensures a

similar temperature curve over instances with different costs and optimization runs with varying maximum iterations.

$$sa\_cr = sa\_tp^{\left(\frac{1}{I_{max} \times sa\_ti}\right)} \quad (2)$$

3.1.3 Adaptive Layer

The centerpiece of ALNS is the *adaptive layer*. It consists of two elements: the operator selection and the weight adjustment.

The operator selection phase chooses the destroy- and repair operators for the upcoming iteration independently of each other. It is implemented as a roulette wheel selection, with each operator's current weight w_o governing the probability p_o to be chosen. Since the selections for repair- and destroy operators are independent, the probability for a given repair operator o^+ calculates with $p_{o^+} = \frac{w_{o^+}}{\sum_o^{\Omega^+} w_o}$.

Adjusting the operator weights can be done in a multitude of ways. Most implementations sampled by Turkeš et al. (2021) follow close to the original proposal by Pisinger and Ropke (2007). Weights are updated after a batch of iterations. During each batch b, operators accumulate a score $\Phi_{o,b}$ dependent on the outcomes of iterations they were active in. We discern between five different outcomes ρ dependent on the cost of the new-found solution and the SA temperature: First: better than the currently best-known solution. Second: better than the currently accepted solution. Third: worse than the currently accepted solution but accepted due to simulated annealing. Fourth: rejected by simulated annealing. Fifth: identical to a solution found in a previous iteration. Each outcome is assigned a reward value ϕ_ρ, which is added to the operator score. Furthermore, the amount of times an operator is chosen within the current batch is tracked as $\Theta_{o,b}$, and the runtime of the selected operator is stored as $t_{o,i}$. We store a hash of each visited solution to be able to compare whether a solution is new.

We initialize all operator weights with $w = 10$ to ensure exploration of the available operators. After each batch, a weight update is performed using Formula (3). It is guaranteed that operators can never entirely leave the roulette selection by increasing the batch reward to at least one (Gullhav et al., 2017). The time factor $tf_{o,b}$ is calculated by Formula (4) where $\varnothing T$ is the average runtime of all operators so far (separated between repair and destroy). ω is the Time-Sensitivity Factor governing how sensitive the weight update is to longer operator runtimes. The minimizing term ensures that operators can only be punished for extended runtimes. Otherwise, operators could increase their weight simply by being fast without finding any improving solutions. Compared to the literature review in section 2.2.2, our approach tracks closest to Gullhav et al. (2017). However, we only calculate and apply tf at the end of a batch instead of after each iteration. This is done to average the runtimes of each operator.

$$w_{o,b} = (1-\alpha) \times w_{o,b-1} + \alpha \times max(1, \frac{\Phi_{o,b} \times tf_{o,b}}{\Theta_{o,b}}) \quad (3)$$

$$tf_{o,b} = min(1, \frac{\Theta_{o,b} \times \varnothing T}{\sum_{i=1}^{\beta} t_{o,i} \times \omega}) \quad (4)$$

Runtime Approximation

Timing the duration of a program operation is non-trivial. Simply using a function akin to a stopwatch to time from the start to the end of an operation is subject to noise, for example, another program slowing down the hardware an algorithm is run on. This noise also makes running a program multiple times with the same seed for random operations impossible, as operators' weights will fluctuate due to slightly different measured runtimes. As shown in section 1, some authors use the CPU cycle time to calculate the operator runtimes (Gullhav et al., 2017; Laborie & Godard, 2007). However, this is not possible for the environment in which our algorithm is run (Python on Windows). Instead, we implement a proxy: the runtime approximation (RTA) score. For this, the algorithm is run multiple times in a profiling mode, giving average runtimes for operators (e.g., *random_destroy*) and sub-operators (e.g., testing the feasibility of a route). From there, an RTA score is assigned to each relevant (sub-)operation. Additionally, some inner loops within the operators are given RTA scores to keep approximations close over wide ranges of customer node amounts. The individual scores are added whenever any function with an RTA score is called and stored as $t_{o,i}$ at the end of the destroy- or repair phase. While the approximations are not 100% accurate, this does not influence the study results. Both the ALNS and the LLNS are only aware of the RTA scores, and the final evaluation criterion is also based on RTA. This approach also allows us to artificially alter operators' efficiency by changing their rta components: We do this in the case of *sorted_repair* to distinguish its efficiency further from the similar *locost_repair*. Figure 2 compares approximated and actual runtimes (in nanoseconds).

Figure 4 in Section 4.3 shows how measured and approximated runtimes behave with growing instance size.

3.2 Learned Large Neighborhood Search

This section explains the setup for training our LLNS policy. We introduce the supplied features, explain potential reward functions, and discuss important hyper-parameters.

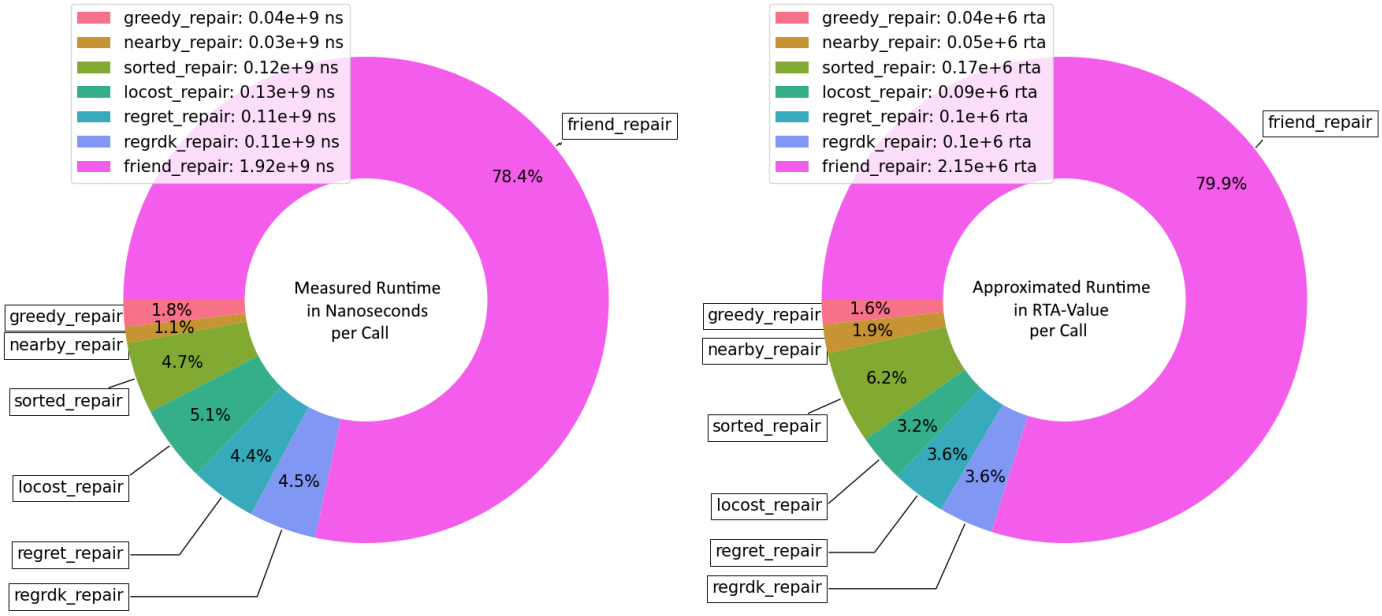

Figure 2: Operator Runtime Comparison: Average Measured Runtime in Nanoseconds and Calculated Approximated Runtime per Call. 90 (¬)ALNS Evaluation Runs, 143-351 Nodes.

3.2.1 Features

One advantage of a trained agent over the adaptive layer is that it can take in much more varied information. The adaptive layers' decisions are only informed by the previous performance of the operators within the current optimization run. An agent can not only utilize information from past runs but can also be fed more data about the current state of the optimization problem. Our goal for the feature selection is to prefer domain-agnostic information to be able to transfer the approach to different problem settings.

Initial testing was done on a small subset of observations, only including the four first features (*completion_percentage, destroy_operator, number_of_destroyed_nodes, iterations_since_last_best*) of the following list. As further discussed in the results in Section 4.5, this proves to be too little information to outperform a handpicked pre-selection of operators. Consequently, we add six additional features:

- **completion_percentage** is the ratio of the already completed iterations to the maximum number of iterations. It is presented as an integer between zero and 100.
- **destroy_operator** informs which destroy operator was used in the current iteration. Since the destroy operators might create sets of non-serviced nodes with different characteristics, some repair operators might perform better after a certain destroy operator was called. The agent has no information on the actual behavior of the destroy operators; it is only provided an index from the list of destroy operators.
- **number_of_destroyed_nodes** carries two insights: first, it can enable choosing repair operators that perform better with more or less destroyed nodes. The RTA of the repair operators is highly sensitive to this number since some operator's RTA scales linearly (e.g., *greedy_repair*), whereas other operators need to repeat calculations for all remaining nodes whenever a node is inserted (e.g., *regret_repair*). The second insight is information on the number of nodes in the problem setting. Since destroy operators have fixed degrees of destruction, knowing both the destroy operator and the number of destroyed nodes tells the agent the size of the problem. As the agent is trained on instances between 100 and 399 nodes, it is possible to learn which operators perform better in smaller or larger instances. Because the training set (see section 4.1) includes instances with a distinct number of nodes, and we do not want the agent to over-train to a point where it optimizes based on the characteristics of the exact instance, a random noise-term of [-1, 0, +1] is added to this observation.
- **iterations_since_last_best** provides information on how long the algorithm has been stuck without finding a new best solution. It ignores solutions accepted only by simulated annealing. Its goal is to enable the agent to choose operators that facilitate either exploration (after no improvements have been found for a long time) or exploration (of new-found improvements). This observation is presented as a percentage of the maximum iterations.
- **current_rta_sum** informs the agent of how much time has been spent in the run so far. As it is a direct part of the reward function (7), this observation is provided to enable the agent to better estimate expected rewards.
- **solution_change_best_vs_initial** is used as a proxy for

the second part of the reward function: the GTO. As we cannot expect to know an optimal solution for any problems outside the training set, we inform the agent of how much it has managed to improve the initial solution. Because of diminishing returns in the search process, the growth of this observation should slow at a similar rate as the convergence of the solution towards the optimum.

- **rta_last_iteration** informs the agent about the duration of the last repair operation.
- **solution_change_last_iteration** informs the agent about the change in objective value within the last iteration. It is calculated as $\frac{c_S}{c_{Sa}}$ where c is the objective value.
- **acceptance_check_result** presents a value between zero and four, corresponding to the SA outcome (new_best / accepted / annealed / rejected / already_visited) of the last iteration.
- **time_complexity_last_operator** groups the repair operators according to their runtime behavior. Figure 4 shows operator runtimes in relation to instance size. Group 1: *greedy_, nearby_*. Group 2: *regret_, regrdk_, locost_*. Group 3: *sorted_*. Group 4: *friend_repair*. This feature is intended to speed up the learning progress in regards to the operator runtime behavior.

3.2.2 Reward Function

Table 3 shows that different reward functions are feasible for LLNS. We identify three approaches: (Reijnen et al., 2022) and Boualamia et al. (2023) give a reward in each iteration with the score ϕ_ρ dependent on the outcome ρ of the acceptance check. This copies the weight adjustments from ALNS. (Bongiovanni et al., 2022) give a reward in each iteration dependent on the objective value change within the last iteration. (Johnn et al., 2023) reward only once, in the last iteration, dependent on the objective value change between the initial and final solution.

We test approaches similar to the three mentioned and find arguments for and against each. As we maximize rewards, reward values for time spent are negative. We use a discount rate of 1, which means future rewards are fully calculated into every previous step within a run:

- Copying the **ALNS reward scheme** as shown in Formula (5) requires no additional work if ALNS is already implemented. Furthermore, compared to the following two approaches, it does not require knowledge of a lower bound solution value for the training dataset. However, it does not distinguish between small and big improvements. A well-trained policy might optimize rewards gained by finding as many small improvements as possible instead of searching for the best final result.

$$R_i = \phi_{\rho_i} \tag{5}$$

- Giving a **single reward in the final iteration** as shown in Formula (6) is straightforward and enables lining up the reward with the algorithm comparison criteria, in our case the expected runtime approximation value at 5% gap to optimum (eRTA-GTO5) explained in Section 4.2. The reward value is the negative cumulative RTA whenever eRTA-GTO5 was first reached, or the expected eRTA-GTO5 if 5% gap to optimum (GTO5) was not reached. This approach has the downside of also rewarding any actions after GTO5 is reached. The policy might make suboptimal decisions after this point, and these actions - which had no impact on the final result - will gain the same rewards as the relevant actions before reaching the GTO-target. This is a downside to our decision to run fixed iterations instead of a fixed runtime (compare Section 2.2.1) and could be mitigated by running for a fixed time and then rewarding based on the reached solution quality.

$$R_i = \begin{cases} 0 & \text{if } i \neq I_{max} \\ -1 \times \text{eRTA-GTO5} & \text{if } i = I_{max} \end{cases} \tag{6}$$

- As we want to optimize for both runtime and solution quality, giving a **reward in each iteration** forces us to consider both the current runtime and the current solution quality. Therefore, we need a formula that can balance the trade-off between runtime and solution quality. We manage this by multiplying the current cumulative RTA with the currently reached GTO+1 $= (\frac{c_{S^*}}{c_{S^{opt.}}})$. As both sides of the multiplication scale differently, we need a balancing factor, which we call the gap-factor gf. This factor must be tuned to achieve the correct trade-off between fast runtimes and good solution quality (see Section 4.4). This complicated approach has the upside of correctly rewarding actions even after the desired GTO-target is reached. Formula (7) shows our reward function for rewards R in each iteration.

$$R_i = -1 \times (\frac{c_{S^*}}{c_{S^{opt.}}})^{gf} \times rta\_sum_i \tag{7}$$

If no lower bounds are known for the training data set, the objective value improvement between the initial and rewarded solution can be used as a proxy for the GTO (Johnn et al., 2023). A short discussion of the performance of the three reward schemes can be found in 4.5.

3.2.3 Training Setup

The training setup for the LLNS agent is built to rely on publicly available libraries. We train a Deep-Q Network (DQN) using TensorFlow's *tf_agents* package (Developers, 2023). We test both the standard DQN-agent and the C51-agent, which calculates a Q-value-distribution instead of a single Q-value (Bellemare et al., 2017). As the outcome of each iteration is heavily stochastic, the distribution might aid the training, but we find no significant differences between DQN- and C51-agents for our application. We implement a replay buffer with a size of 10% of all training steps using the *tf_uniform_replay_buffer* package (Developers, 2023). Extensive testing is needed to find a learning

rate that neither gets stuck in a local optimum nor cannot keep any learned information. We find that a decaying learning rate works best. Special attention is needed for the epsilon-greedy factor ϵ, which determines how often a random action should be taken during training. As every step influences both the solution quality and the runtime, too many random actions can prohibit the learning of strong policies. We choose a decaying ϵ.

We build our ANN with three hidden layers of size [256, 128, 64], following the theoretical advice by Stathakis (2009) and Bengio (2012), as well as the LLNS-implementation by Boualamia et al. (2023). Training runs featured up to two million transitions, calculated live using a set of training instances with between 101 and 393 nodes. This leads to training runs between 24 and 92 hours. Since the majority of training time is spent solving the routing problem, we believe that significant time could be saved by switching to learning in an offline transition database.

Section 4.5 explains our learnings from the training runs. Appendix Table A3 lists all hyper-parameter values.

4 Results

The following Chapter presents our results for analyzing the performance of our proposed extensions to the ALNS. We start by outlining the comparison setting by explaining the instance datasets (Section 4.1), the multi-objective comparison criterion (Section 4.2), and by presenting a quantitative analysis of the repair operator performance (Section 4.3).

We then present the results of our meta-parameter tuning (Section 4.4) and report on the behavior of the LLNS training runs with varying hyper-parameters, feature sets, and reward functions (Section 4.5).

Section 4.6.1 presents the main results of our study, the comparison of ALNS, (¬)ALNS, tALNS and LLNS. Subsections 4.6.2 and 4.6.3 present additional results for the behavior of tALNS with different operator portfolios, as well as a test of the generalization capabilities of tALNS and LLNS in larger instances (700+ nodes).

To reduce variability, we only use *random_repair* with $dod = 0.33$ for our training- and evaluation-runs. Testing for impact on the eRTA-GTO5 of the tALNS implementation, we found only a non-significant change (-1.37% average eRTA-GTO5, p-value 0.888) compared to using all destroy operators.

4.1 Instance Dataset

The publicly available CVRP-library by Uchoa et al. (2017) is used to generate problem instances. It is split into a training set and an evaluation set, with the evaluation set containing three files for each century of customer nodes. In this context, *century* refers to a group of instances with the same hundredths place value in their number of nodes, e.g., 100 - 199 nodes are grouped into century 100. To test the algorithms under varying conditions, we select test instances with varying characteristics of depot position, customer clustering, and minimum vehicles needed. Appendix Table A1 gives detailed information on the evaluation instances. A plot for each instance can be found on the CVRPLIB website (Lima et al., 2023). The reward functions for LLNS training and the algorithm comparison criterion rely on a gap to optimum (GTO) value. (Uchoa et al., 2017) provide the best-known solution with each of their problem instances, which, for most instances, is the optimum solution proven by exact solving. The best-known solution is used as a stand-in if the optimal solution has not been verified.

4.2 Algorithm Comparison Criterion: Runtime at Fixed Solution Quality

As shown in section 2.2.3, a straightforward method to judge different meta-heuristics for both speed and performance is to compare the solution quality after a fixed runtime is reached. Due to multiple reasons, we decide against this approach: Firstly, SA lowers the temperature after each iteration. Therefore, not using fixed iteration targets leads to unpredictable temperature behavior. Secondly, fixing iterations allows for a degree of automatic adaption of runtimes to problem size. As the degree of destruction of each destroy-operator is a fixed percentage of the instance nodes, each iteration's neighborhood search grows in size when more nodes are considered. These larger sub-problems take more time to solve, leading to overall longer solving times for larger instances. This creates a need to consider different runtime-stopping-criterions dependent on instance size, increasing complexity. Section 4.3 explores the runtime of repair-operators in relation to instance size. Lastly, within training, the agent is rewarded after each iteration, with the outcomes of later iterations influencing early decision-making via discounted rewards. An unpredictable number of future rewards due to fluctuating iterations adds an additional undesirable variable to the training process. For these reasons, we decide to keep the fixed iterations also used by Pisinger and Ropke (2007).

In the sampled literature, none of the authors using an iteration-based stopping criterion implements a quantitative approach to judge algorithms on speed and solution quality. Instead, both measures are reported and compared independently (Wu et al., 2017; Zhang & Yang, 2021). Gullhav et al. (2017) stop after fixed iterations but judge solution quality at three fixed durations. This is not feasible for our approach, as the diverse problem instances and operator runtimes lead to massive differences in solving times. For example, our ALNS capped at 2000 iterations can solve a simple layout with 150 customers to optimality after 30 seconds but can take up to 30 minutes to reach a 5% GTO in a very large instance with small transport capacity per vehicle.

A lower outcome variability is observed for the final GTOs. In 180 runs of ALNS in the evaluation dataset, no run exceeds 9% GTO, and the standard deviation of GTOs is measured at 0.022 (see Table 7). Therefore, we decide to use the RTA needed to get to GTO5 as the comparison criterion. For runs that do not reach GTO5, an expected runtime

approximation value at 5% gap to optimum (eRTA-GTO5) is calculated. Algorithm 2 shows how the expected runtime approximation (eRTA) is calculated using the expectation factors shown in Table 4. The eRTA factors are derived by running 300 instances for 10,000 iterations each. The ratios between the median RTAs at each relevant GTO-milestone are calculated for centuries 100, 200, and 300. Instances with more than 399 nodes use the factors of century 300.

Runs that do not reach GTO10 are considered as *failed* and are punished with the highest eRTA-GTO5 out of any compared algorithm runs on the given problem instance. Algorithm results that include runs which do not reach GTO10 will always be clearly marked as such. All runs within the main algorithm comparison in Table 7 reach GTO10.

Table 4: Expectation Factors for Runtimes at Gap to Optimum Values

Century	0.10 → 0.09	0.09 → 0.08	0.08 → 0.07	0.07 → 0.06	0.06 → 0.05
100	1.63	1.34	1.78	1.38	1.37
200	1.62	1.87	1.72	1.60	1.24
300	2.25	1.87	1.75	1.26	1.48

Century refers to the instance size, with all instances between 100 and 199 nodes being sorted into century 100.

Cell values display the factors for estimating the time needed to reach a smaller gap to optimum.

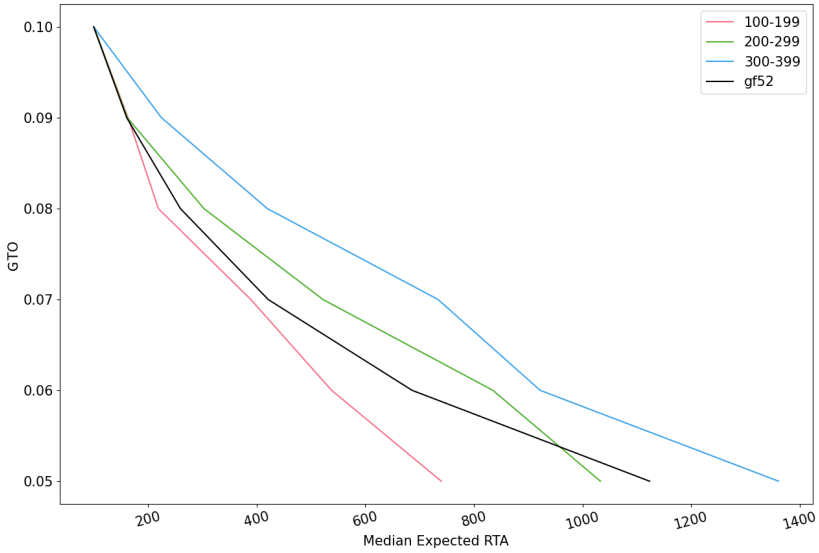

Figure 3: Colored: Curves for the Expected RTA, starting at RTA-GTO10=100, by Instance Size Century. Black: Curve for the Gap Factor gf=52.

Algorithm 2: Algorithm to Calculate eRTA for a given Century

Inputs: observed_rta_at_gto, eRTA_gto_factors[century]
rta_steps = [0.09, 0.08, 0.07, 0.06, 0.05]
if *observed_rta_at_gto[0.10] == 'not_reached'* **then**
  return '10%_not_reached'
else
  eRTA_at_gto[0.10] = observed_rta_at_gto[0.10]
for *gto in rta_steps* **do**
  if *observed_rta_at_gto[gto] != 'not_reached'* **then**
    eRTA_at_gto[gto] = observed_rta_at_gto[gto]
  else
    eRTA_at_gto[gto] = eRTA_at_gto[$gto + 0.01$] * eRTA_gto_factors[$gto + 0.01$]
return eRTA_at_gto

4.3 Repair Operator Comparison

Figure 4 shows repair operator runtime behavior in growing instances. We observe that the RTA-approach does not accurately model the growing runtimes in larger instances. Furthermore, measured nano-second (ns) runtimes peak in instances with the depot in a corner, while the calculated RTA spikes with higher vehicle counts. This points towards skewed RTA-values for some suboperations. Section 3.1.3 explains why this does not invalidate our algorithm comparisons. See Appendix Table A1 for an overview of instance layouts. Appendix Table A2 shows mean and median operator runtimes for all instances up to 600 nodes.

Nevertheless, in both ns- and RTA-runtimes, it is clear that operator runtimes differ significantly, with an average call of *friend_repair* taking more than 50 times longer than *nearby_repair*. The classic operators like *greedy_repair* and *regret_repair*, which are routinely used in ALNS-literature,

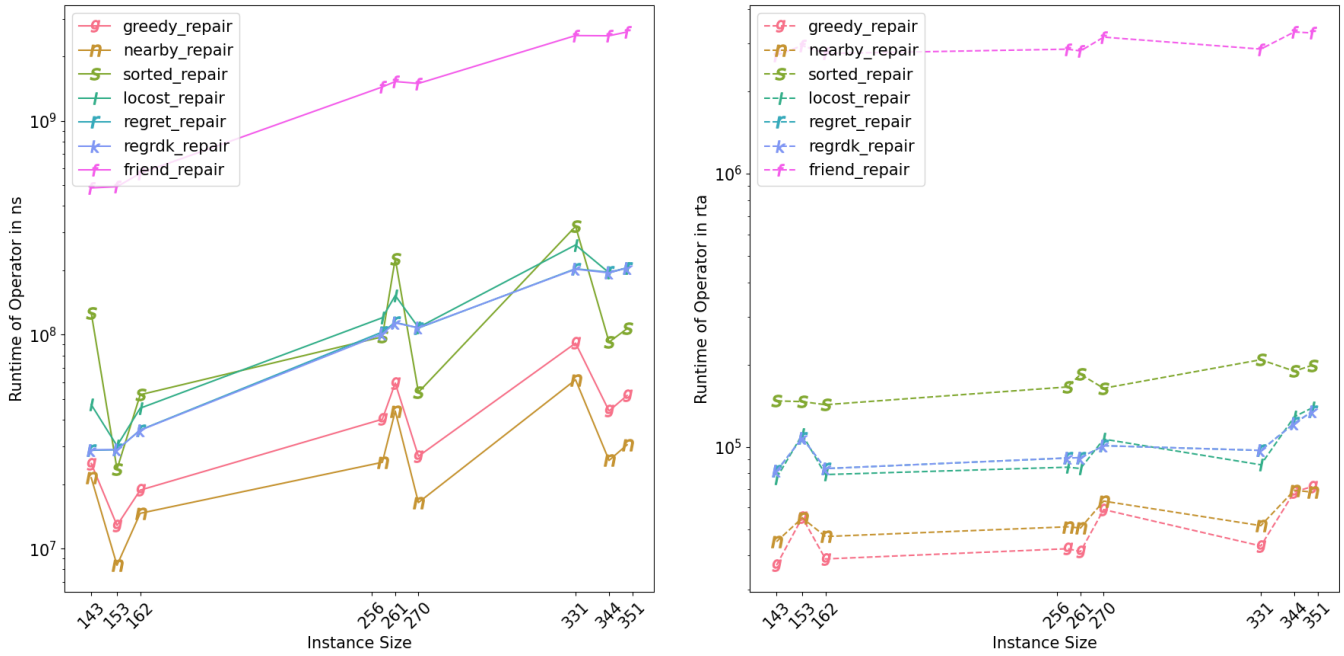

Figure 4: Operator Runtime in Observed Nanoseconds (left) and Calculated RTA (right). 10 (¬)ALNS runs per instance. Logarithmic y-scale.

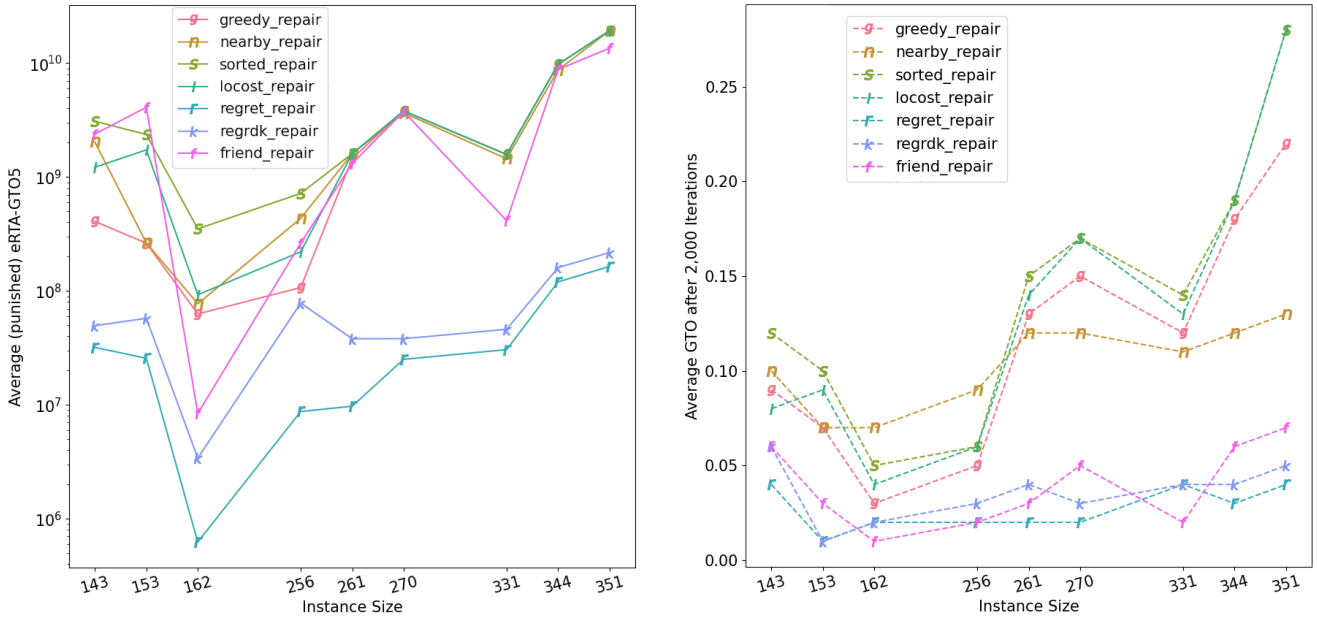

Figure 5: eRTA-GTO5 (left) and GTO after 2000 Iterations (right) reached by Runs with Single Operator Portfolios. Logarithmic y-scale for left plot. 10 runs per instance and operator.

still differ by a factor of 10 (Windras Mara et al., 2022). Profiling runs show that for our algorithm implementation, repair operators take up between 75% and 95% of the metaheuristic's runtime, the remainder being split between the destroy phase and miscellaneous operations like weight adjustments or acceptance checks. These timing results clearly motivate the need for a time-sensitive operator weight adjustment scheme.

Figure 5 displays operator performance in Single Operator Portfolio runs. This mimics testing a LNS for each operator. The findings show that *regret_repair* has the highest efficiency, *friend_repair* finds good solutions but is comparatively slow, and the fastest four operators are unable to reach GTO10 in larger instances.

In addition to the LNS-results, we analyze operator performance in (¬)ALNS. These results are less conclusive and

Table 5: Meta-Parameter Tuning Values

Parameter	(¬)ALNS	ALNS	tALNS	LLNS	Range	Value
ϕ_ρ	X	✓	✓	X	[20-100, 5-50, 5, 1, 1]	[50, 10, 5, 1, 1]
ω	X	X	✓	X	0.1 - 1.5	1.20 (all operators) 0.15 (pre-selected*)
β	X	✓	✓	X	10 - 100	50
α	X	✓	✓	X	0.1 - 0.9	0.5
dod	✓	✓	✓	✓	0.1 - 0.5	0.33
gf	X	X	X	✓	not applicable	52

ϕ_ρ: Acceptance Reward for Result ρ. ω: Time Sensitivity Factor. β: Batch Size. α: Weight Update Rate.
dod: Degree of Destruction for Destroy Operators. gf: Gap Factor for LLNS-rewards.
*__pre-selected__ refers to a reduced operator portfolio of *greedy_, nearby_, locost_* and *random_repair*

can be found in Appendix Section A5.

4.4 Meta-Parameter Tuning

ALNS contains several parameters that need tuning for the algorithm to perform optimally (Ropke & Pisinger, 2006). We will refer to them as meta-parameters to distinguish them from parameters that influence Reinforcement Learning (RL), often called hyper-parameters. As the objective of this study is to compare different versions (ALNS, (¬)ALNS, tALNS, LLNS) of the same meta-heuristic implementation, we reduce complexity by only tuning meta-parameters which influence the performance of a subset of the compared algorithms. Therefore, we leave, for example, SA-parameters untuned, as they do not impact the comparison of our ALNS-variants. We use starting values from the existing literature for these non-tuned parameters and make slight adjustments by hand. Appendix Table A3 gives a full list of all final parameter values.

As meta-parameter values influence each other's best value, evaluations must be conducted for each combination of values in the testing ranges. We reduce complexity by forming smaller clusters of meta-parameters. The clusters are then optimized in sequence. We group the acceptance check rewards ϕ with the time-sensitivity factor ω as both directly affect the weight update values. Furthermore, we group the batch size β with the weight update rate α, as their combination controls how fast weights can fluctuate. Table 5 lists the tuned parameters and which algorithm variants they affect.

A challenge of tuning the meta-parameters is dealing with different instance sizes: Tuning is done on instances between 101 and 393 nodes (the training set for LLNS training). The optimized objective value is the eRTA-GTO5. Since this value scales with instance size and layout, the standard deviation of tuning results over a wide span of instances tends to be very high. Therefore, reaching a tuning result with low p-values requires excessive runs. Filtering by instance century reveals that different meta-parameter values excel for different centuries (and most likely also other problem characteristics like the instance layout). P-values reach significantly smaller values (<0.05) when analyzing century averages instead of averages over all tuning instances. Nevertheless, a single meta-parameter value has to be chosen for all centuries. If different values are optimal for different instance size centuries, we opt for the best value for century 200.

For the LLNS, we only need to tune one meta-parameter, the gap factor gf. The hyper-parameter selection for RL training is explained in Section 4.5. The gap factor determines how expired runtime and reached GTO are weighted against each other in Formula (7). It balances the trade-off between fast runtimes and good solution quality. We already calculate the expected runtime behavior in relation to GTO-reached for the eRTA-factors. We determine the gf by matching it to the eRTA-curve for century 200 using Formula (8). Figure 3 shows the eRTA-curves and a curve for $gf = 53$.

In Formula (8), $eRTAf_{0.10\rightarrow gtd}$ refers to the compounding eRTA-factors (of century 200) between GTO10 and the GTO-milestones we are summing over. Table 4 lists milestones and eRTA-factors. We chose $gf \varepsilon \mathbb{N}^+$ so that the squared error between the GTO- and the RTA-parts of the formula is minimized. This ensures that the Reward Formula (7) produces similar rewards for any points (RTA/GTO) on the eRTA-curve for century 200.

$$minimize \sum_{gto=0.05}^{0.10} \left(\frac{1.1^{gf}}{(1+gto)^{gf}} - eRTAf_{0.10\rightarrow gto}\right)^2 \quad (8)$$

4.5 Q-Learning Training Results

As stated in Section 3.2.3, we test multiple different learning rates, ϵ-greedy values, feature portfolios, and reward functions for our LLNS-training setup. A statistical analysis of the performance of given combinations of hyper-parameters and environment setups is out of the scope of this study. This section will nevertheless give a short review of our training results.

After tuning by hand, we settle on a decaying ϵ for ϵ-greedy, starting at 80% and lowering over the first 30% of episodes to 0.1%. The learning rate starts at $1e^{-}4$ and decays to $1e^{-}6$ over the first 75% of episodes.

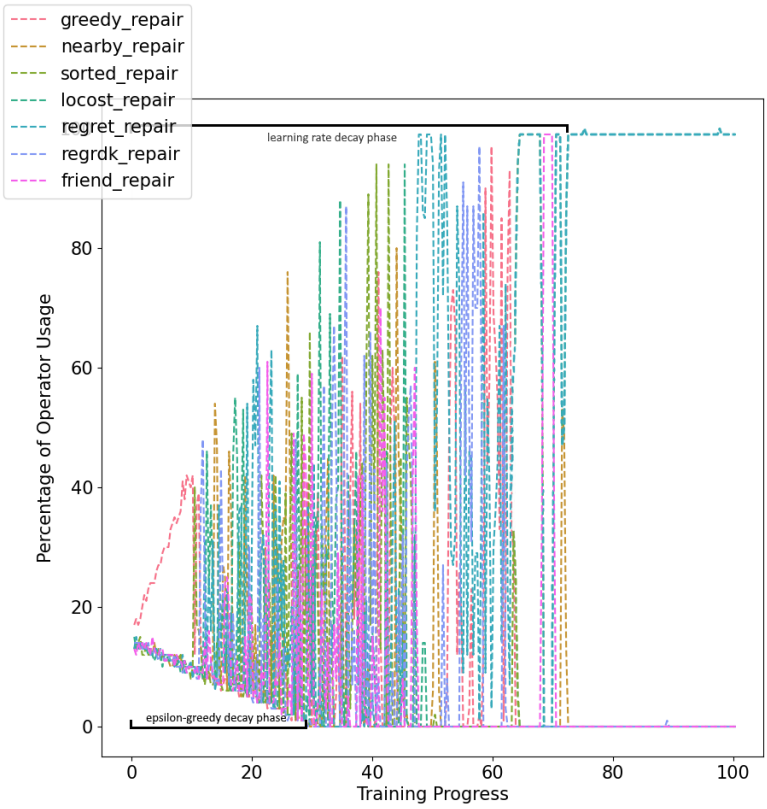

Figure 6: Operator Selection Percentage during Training. Section 0-30 shows decaying ϵ. Section 75-100 shows fully decayed Learning Rate.

Figure 6 shows the policy behavior during a training run. The decaying ϵ-greedy is visible between 0% and 30% training progress. After 75% progress, we observe fewer changes in the policy, which is caused by the decayed learning rate. The policy converges towards choosing *regret_repair* in almost all iterations.

Out of the three tested reward schemes discussed in Section 3.2.2, we do not converge to a strong policy within a reasonable training time using ALNS-rewards. Both the multi-objective rewards in each iteration as well as the single eRTA-GTO5-reward in the final iteration reach policies relying solely on *regret_repair*, with the single final reward scheme doing so with slightly higher consistency.

Furthermore, we test two different sets of features. The small set (first four features mentioned in 3.2.1) does not converge to *regret_repair* without decaying ϵ-greedy and learning rate. The large feature portfolio does so, but only after long training times. With the decaying hyper-parameters implemented, both feature portfolios converge to using only *regret_repair*.

Whether the trained selection policies convergence towards single-operator usage is caused by the overwhelming effectiveness of the chosen operator (potentially exacerbated by the RTA-calculations of the operators) or by insufficient training is subject to further research. For now, we can show that LLNS is able to select the most efficient operator, effectively turning ALNS back into LNS, but without the need for hand-picking an operator. We cannot prove that no better policy utilizing more repair operators exists. We test the training setup with *regret_repair* and *regrdk_repair* removed, but are unable to learn strong policies for this setup. Further research is needed to investigate whether this issue is caused by unfit hyper-parameters for the smaller operator portfolio, insufficient training times, flaws in our general training setup, or other unknown causes.

4.6 Algorithm Comparison

This section presents and discusses our main algorithm comparison results. Furthermore, we present additional tests for tALNS-behavior under different operator portfolios as well as the generalization capabilities of our algorithms.

4.6.1 Algorithm Comparison Results

Figure 7 shows which repair operators are preferred by different algorithms. Two operator portfolios are present: The full portfolio of all seven repair operators, and a pre-selected (PS) portfolio that simulates an algorithm designer hand-picking promising operators for a given problem domain. We can observe how the average eRTA-GTO5 decreases with declining usage of *friend_repair* and increasing usage of *regret_repair*. We can also see the impact of a

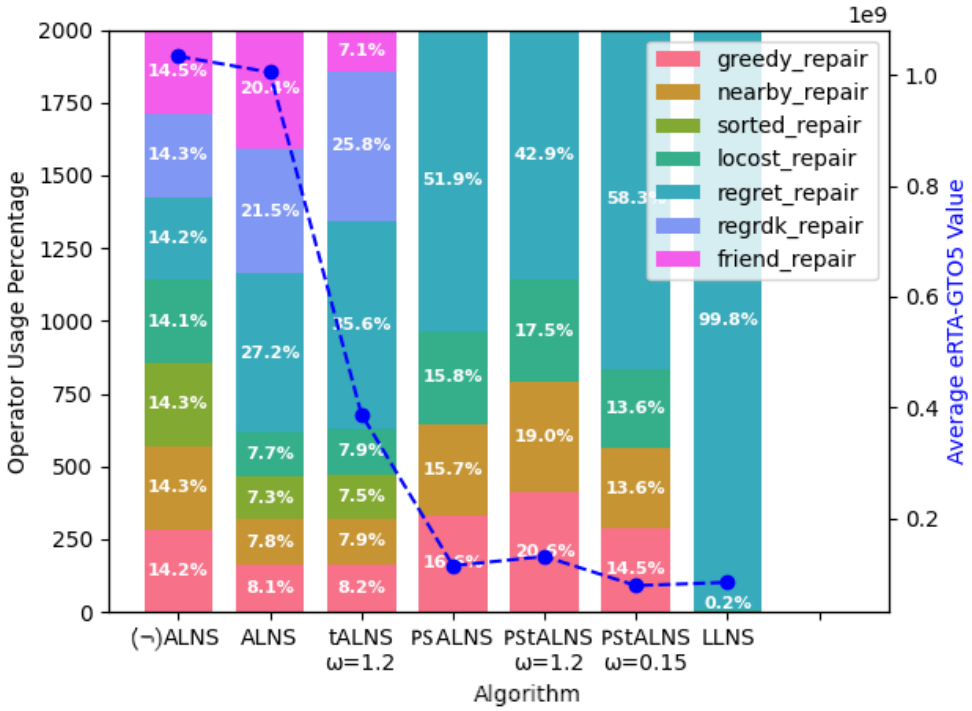

Figure 7: Operator Selection Percentage of Compared Algorithms as Stacked Bars. Overlaid is the average eRTA-GTO5 of the algorithms in blue. LLNS chooses *locost_repair* in 0.2% of iterations.
ps refers to operator pre-selection: *greedy_*, *nearby_*, *locost_* and *random_repair*.
ω is the Time-Sensitivity Factor.

Table 6: Algorithm Performance as Matrix Comparison

	(¬)ALNS	ALNS	tALNS, $\omega = 1.2$	PS ALNS	PS tALNS, $\omega = 1.2$	hp tALNS, $\omega = 0.15$
ALNS	-2.77%, p=0.7026					
tALNS, $\omega = 1.2$	-62.49%, p=0.0000	-62.49%, p=0.0000				
PS ALNS	-88.94%, p=0.0000	-88.94%, p=0.0000	-70.50%, p=0.0000			
PS tALNS, $\omega = 1.2$	-87.32%, p=0.0000	-87.32%, p=0.0000	-66.19%, p=0.0000	+14.61%, p=0.0326		
PS tALNS, $\omega = 0.15$	-92.42%, p=0.0000	-92.20%, p=0.0000	-79.89%, p=0.0000	-31.48%, p=0.0000	-40.22%, p=0.0000	
LLNS	-91.85%, p=0.0000	-91.85%, p=0.0000	-78.27%, p=0.0000	-26.32%, p=0.0079	-35.71%, p=0.0001	+7.53%, p=0.5954

Algorithm in row compared to Algorithm in column. Percentage values calculated as average eRTA-GTO5 of $\frac{row}{column} - 1$.
p-values are two-sided. ω is the Time Sensitivity Factor. **PS** refers to operator pre-selection.

correctly tuned sensitivity factor ω for the time-sensitive ALNS, depending on the operator portfolio. This topic is further explained in Section 4.6.2.

As the differences between our proposed algorithms and classic ALNS shown in Table 7 are too large to provide meaningful p-values, we display a direct comparison between each algorithm in Table 6.

Table 7 shows the average eRTA-GTO5 as the main algorithm comparison criterion. As the trade-off between fast runtimes and good solution quality might change depending on problem scenarios, we also report the final GTO and RTA after 2000 iterations. Lastly, the average iteration of the best solution is reported. This value allows us to estimate whether adding additional iterations would lead to better solution quality, i.e., it tells us if there is room for improvement left at the end of the 2000 iterations.

Table 7: Results of Algorithm Performance Comparison

Instance	Avg. eRTA-GTO5	Std. Dev. eRTA-GTO5	Avg. Final GTO	Avg. Final RTA	Avg. Iter. of Best Solution	% Change to ALNS	T-Value	p-Value***
				(¬)ALNS				
143-k7-corner	4.96E+8	1.10E+8	2.50%	9.12E+8	1857	-1.67%	-0.24	0.8143
153-k22-clustr	6.81E+8	1.16E+8	2.60%	10.18E+8	1861	+20.92%	3.20	0.0108
162-k11-center	2.04E+8	1.82E+8	0.90%	9.39E+8	1779	+128.42%	1.99	0.0773
256-k16-clustr	3.21E+8	1.13E+8	1.60%	9.57E+8	1735	+33.06%	2.24	0.0521
261-k13-corner	6.14E+8	1.53E+8	3.30%	9.90E+8	1903	-8.58%	-1.19	0.2653
270-k35-center	11.91E+8	3.71E+8	5.20%	10.85E+8	1853	+9.97%	0.92	0.3814
331-k15-corner	5.85E+8	1.53E+8	3.60%	10.07E+8	1659	-22.97%	-3.59	0.0058
344-k43-center	20.65E+8	5.92E+8	6.70%	11.42E+8	1746	-7.31%	-0.87	0.4066
351-k40-clustr	31.46E+8	9.47E+8	7.40%	11.57E+8	1788	+8.29%	0.80	0.4420
Average*	10.34E+8	9.99E+8	3.76%	10.23E+8	1798	+2.85%	0.27	0.7864
				ALNS				
143-k7-corner	5.05E+8	2.92E+8	2.25%	13.63E+8	1755			
153-k22-clustr	5.63E+8	1.63E+8	1.80%	10.21E+8	1752			
162-k11-center	0.89E+8	0.87E+8	0.50%	13.12E+8	1674			
256-k16-clustr	2.42E+8	1.17E+8	1.25%	15.28E+8	1672			
261-k13-corner	6.72E+8	2.70E+8	2.60%	13.14E+8	1763			
270-k35-center	10.83E+8	2.88E+8	4.50%	13.42E+8	1793			
331-k15-corner	7.59E+8	3.69E+8	3.30%	16.13E+8	1732			
344-k43-center	22.28E+8	8.50E+8	5.95%	14.45E+8	1824			
351-k40-clustr	29.05E+8	9.12E+8	6.85%	14.84E+8	1854			
Average**	10.05E+8	10.05E+8	3.22%	13.80E+8	1758			
				tALNS, ω=1.2				
143-k7-corner	1.46E+8	0.60E+8	1.10%	5.46E+8	1585	-71.15%	-18.87	0.0000
153-k22-clustr	2.54E+8	0.61E+8	1.50%	5.40E+8	1904	-54.86%	-16.11	0.0000
162-k11-center	0.56E+8	0.48E+8	0.90%	4.97E+8	1638	-37.35%	-2.20	0.0557
256-k16-clustr	0.82E+8	0.41E+8	1.20%	5.36E+8	1721	-66.05%	-12.22	0.0000
261-k13-corner	1.85E+8	1.11E+8	2.40%	6.36E+8	1481	-72.49%	-13.93	0.0000
270-k35-center	5.96E+8	1.10E+8	4.80%	7.07E+8	1833	-44.98%	-14.03	0.0000
331-k15-corner	2.86E+8	0.72E+8	3.20%	5.79E+8	1784	-62.38%	-20.66	0.0000
344-k43-center	8.67E+8	2.93E+8	6.00%	7.21E+8	1785	-61.08%	-14.68	0.0000
351-k40-clustr	10.18E+8	2.25E+8	7.00%	6.92E+8	1763	-64.96%	-26.52	0.0000
Average*	3.88E+8	3.61E+8	3.12%	6.06E+8	1722	-61.43%	-16.21	0.0000
				Pre-Selected ALNS****				
143-k7-corner	0.68E+8	0.34E+8	2.65%	1.61E+8	1370	-86.56%	-58.24	0.0000
153-k22-clustr	0.59E+8	0.27E+8	1.30%	1.64E+8	1794	-89.47%	-84.49	0.0000
162-k11-center	0.18E+8	0.10E+8	0.85%	1.62E+8	1396	-79.89%	-31.94	0.0000
256-k16-clustr	0.37E+8	0.17E+8	1.90%	2.00E+8	1647	-84.70%	-52.93	0.0000
261-k13-corner	0.41E+8	0.23E+8	2.65%	1.78E+8	1536	-93.84%	-120.83	0.0000
270-k35-center	1.12E+8	0.39E+8	3.75%	2.07E+8	1432	-89.68%	-111.92	0.0000
331-k15-corner	1.47E+8	0.51E+8	3.70%	3.15E+8	1524	-80.60%	-53.66	0.0000
344-k43-center	2.33E+8	0.87E+8	5.25%	2.73E+8	1495	-89.55%	-102.02	0.0000
351-k40-clustr	3.14E+8	0.68E+8	5.45%	2.84E+8	1651	-89.20%	-169.25	0.0000
Average**	1.14E+8	1.05E+8	3.06%	2.16E+8	1538	-88.62%	-113.61	0.0000
				Pre-Selected tALNS, ω=0.15****				
143-k7-corner	0.44E+8	0.28E+8	2.60%	1.40E+8	1551	-91.19%	-51.98	0.0000
153-k22-clustr	0.71E+8	0.16E+8	1.20%	1.84E+8	1797	-87.48%	-97.32	0.0000
162-k11-center	0.07E+8	0.06E+8	1.00%	1.41E+8	1512	-92.31%	-41.05	0.0000
256-k16-clustr	0.22E+8	0.09E+8	1.00%	1.60E+8	1552	-90.83%	-79.05	0.0000
261-k13-corner	0.24E+8	0.20E+8	2.10%	1.58E+8	1757	-96.50%	-104.27	0.0000
270-k35-center	1.26E+8	0.26E+8	3.80%	1.84E+8	1700	-88.39%	-116.60	0.0000
331-k15-corner	0.54E+8	0.42E+8	3.00%	1.66E+8	1690	-92.86%	-52.47	0.0000
344-k43-center	1.68E+8	0.34E+8	4.80%	1.98E+8	1657	-92.45%	-193.58	0.0000
351-k40-clustr	1.90E+8	0.48E+8	5.00%	2.04E+8	1701	-93.48%	-177.11	0.0000
Average*	0.78E+8	0.69E+8	2.72%	1.71E+8	1657	-92.20%	-127.22	0.0000
				LLNS				
143-k7-corner	0.29E+8	0.16E+8	1.40%	1.63E+8	1442	-94.26%	-93.91	0.0000
153-k22-clustr	0.34E+8	0.19E+8	1.10%	2.03E+8	1828	-93.97%	-88.04	0.0000
162-k11-center	0.01E+8	0.00E+8	0.50%	1.67E+8	974	-99.32%	-625.95	0.0000
256-k16-clustr	0.11E+8	0.05E+8	0.90%	1.82E+8	1674	-95.57%	-133.44	0.0000
261-k13-corner	0.15E+8	0.08E+8	2.00%	1.83E+8	1878	-97.71%	-254.27	0.0000
270-k35-center	1.58E+8	0.18E+8	2.80%	2.20E+8	1859	-85.44%	-166.44	0.0000
331-k15-corner	0.15E+8	0.13E+8	1.90%	1.94E+8	1926	-97.98%	-177.80	0.0000
344-k43-center	2.01E+8	0.29E+8	4.00%	2.30E+8	1877	-90.97%	-223.58	0.0000
351-k40-clustr	2.95E+8	0.80E+8	5.30%	2.54E+8	1843	-89.86%	-103.80	0.0000
Average*	0.84E+8	1.05E+8	2.21%	2.00E+8	1700	-91.62%	-83.13	0.0000

*10 runs per Instance, **20 runs per Instance, ***two sided p-Value
****Pre-Selected** (PS) refers to a reduced operator portfolio of *greedy_, nearby_, locost_* and *random_repair*.
ω: Time Sensitivity Factor for tALNS. Higher value means lower rewards for slow operators.
n: instance size in number of customer nodes, **k**: min. amount of vehicles needed, **corner/clustr/center**: instance layout

We can deduct the following insights from the algorithm comparison results:

- Similar to the results found by Turkeš et al. (2021), our ALNS barely outperforms random operator selection (¬)ALNS by 2.85%, with an insignificant p-value of 0.786.
- The tALNS significantly outperforms ALNS: by 62.49% (p=0.000) for the complete operator portfolio, by 31.48% (p=0.000) in the pre-selected operator portfolio. This shows that the time-sensitive ALNS is a strong extension to classic ALNS, especially when operator portfolios are large and include inefficient operators. Section 4.6.2 goes into depth on the influence of the operator portfolio and the importance of including a tuned sensitivity factor ω.
- In this study, we are able to significantly outperform the ALNS with a RL-trained repair operator selection policy. We improve our objective value by 91.85%, reducing runtimes to less than 10% of ALNS. While early versions of our LLNS struggle to reach the results of ALNS with a pre-selected operator portfolio, the addition of decaying hyper-parameters and additional (yet still domain-agnostic) features allows the training to converge towards a LNS-style policy featuring mainly one very efficient operator. It is unclear whether the conditions of our problem setting and operator portfolio or insufficient training cause this result. Nevertheless, we can prove that in our setting, LLNS can replace the time-intensive and error-prone operator preselection by hand.

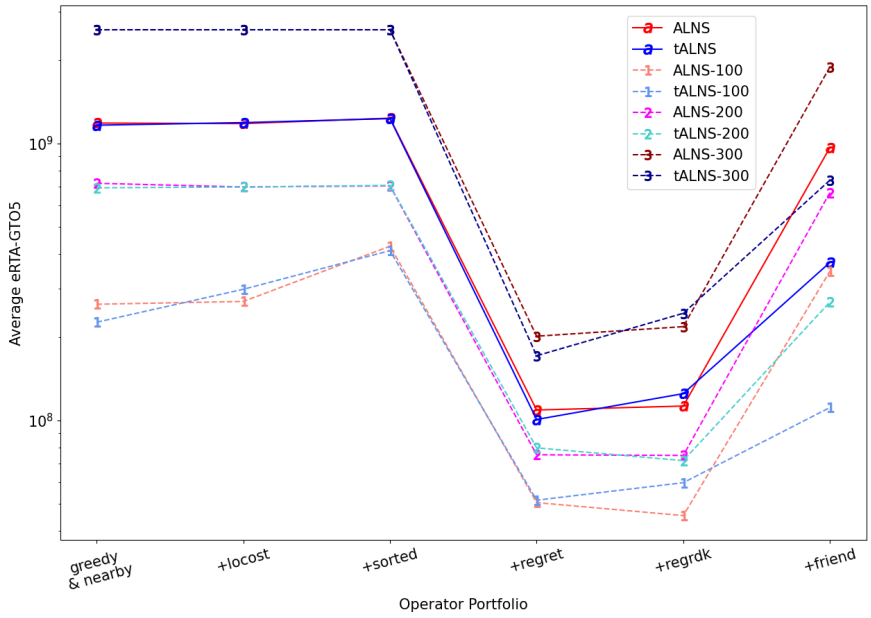

Figure 8: Performance of ALNS and ALNS with growing Operator Portfolios. Time-Sensitivity Factor ω is fixed at 1. 5 runs per algorithm, portfolio and instance.

4.6.2 Time-Sensitive ALNS: Operator Portfolio Comparison

This section compares under which boundary conditions tALNS outperform ALNS given a fixed Time-Sensitivity Factor $\omega = 1$, which is equal to not utilizing a sensitivity factor. The Literature Review Table 1 shows that out of nine sampled studies, only Gullhav et al. (2017) implement a tuneable factor to control the impact of runtime. Figure 8 shows the average eRTA-GTO5 reached by the two algorithms when provided with growing operator portfolios. We order the operators by their average iteration runtime and add a new slowest operator for each new portfolio. The figure shows average objective values over all instance centuries and an additional graph for each evaluation instance size century.

Multiple insights can be gained from this analysis: Firstly, the faster operators (up to *sorted_ repair*) are not strong enough to solve the 300-century instances to GTO10 within 2000 iterations. As no expected RTA can be calculated, we punish these runs with the slowest run from any portfolio that does reach GTO10. The exact punished and unpunished values, as well as a significance test, can be found in Appendix Table A. Secondly, tALNS strongly outperforms ALNS once the very slow *friend_repair* operator is added. This is the optimal scenario for tALNS: the presence of an operator that finds a high amount of improvements per iteration - leading to high ALNS operator weight - but performs poorly in improvements per runtime. Lastly, we notice that tALNS underperforms once the two *regret*-operators are added. This happens when the most effective operators (*regret_* and *regrdk_repair*) are also the comparatively slowest operators. As shown in Section 4.6.1, a LNS with only *regret_repair* outperforms all other algorithms, but tALNS still lowers the rewards of *regret_repair* as it is slower than the average. It therefore takes longer for the weight of the most effective operator to

Table 8: Results of Tuning Time-Sensitivity Factor ω for Pre-Selected Operator Portfolio (*greedy_, nearby_, locost_* and *regret_repair*)

ω:	0.00	0.10	0.15	0.25	0.35	0.50	0.75	1.00	1.20
	1.14E+8	0.94E+8	0.78E+8	0.88E+8	0.92E+8	0.84E+8	0.89E+8	0.92E+8	1.31E+8

ω: Time Sensitivity Factor for tALNS. At minimum 45 runs per parameter value.

Table 9: Results of Testing tALNS and LLNS in Large Instances

Instance**	Avg. eRTA-GTO5	Std. Dev. eRTA-GTO5	Avg. Final GTO	Avg. Final RTA	Avg. Iter. of Best Solution	% Change to ALNS	T-Value	p-Value****
				ALNS				
n701-k44_corner	25.97E+8	5.18E+8	6.4%	18.48E+8	1772			
n716-k35_clustr	27.31E+8	13.72E+8	6.4%	17.18E+8	1688			
n733-k159_center	-*	-*	11.4%	29.07E+8	1921			
Average* **	26.64E+8	9.81E+8	8.13%	21.57E+8	1794			
				tALNS, ω=1.2				
n701-k44_corner	6.19E+8	1.20E+8	6.2%	8.37E+8	1739	-76.15%	-36.80	0.0000
n716-k35_clustr	12.09E+8	3.63E+8	6.8%	7.99E+8	1825	-55.73%	-9.38	0.0007
n733-k159_center	-*	-*	11.4%	15.50E+8	1707	-	-	-
Average* **	9.14E+8	4.02E+8	8.13%	10.62E+8	1757	-65.68%	-16.87	0.0000
				LLNS				
n701-k44_corner	3.23E+8	0.56E+8	5.60%	2.68E+8	1918	-87.56%	-90.76	0.0000
n716-k35_clustr	3.32E+8	0.74E+8	5.20%	2.92E+8	1863	-87.86%	-72.39	0.0000
n733-k159_center	-*	-*	12.00%	10.94E+8	1709	-	-	-
Average* **	3.27E+8	0.62E+8	7.60%	5.52E+8	1830	-87.71%	-145.73	0.0000

*GTO10 was not reached, **5 runs per Instance and Algorithm, ***Runs which reached GTO10, ****two sided p-Value
n: instance size in number of customer nodes, **k**: min. amount of vehicles needed, **corner/clustr/center**: instance layout

outgrow the faster operators, leading to overall longer runtimes. For this analysis it is very important to stress that the time-sensitivity meta-parameter ω is not tuned for each portfolio. If we re-tune for each portfolio, we expect ω to be < 1 for the portfolios where the slowest operator is the most efficient. In fact, a sensitivity value of $\omega = 0$ would behave exactly like ALNS.

While Gullhav et al. (2017) tune their ω only to as low as 0.5, we test ω-values from 0 to 1.2 for the PS operator portfolio (including *greedy_, nearby_, locost_* and *regret_repair*). Table 8 shows the tuning results, and Figure 7 shows that a sensitivity factor of $\omega = 0.15$ not only outperforms ALNS, but matches the performance of LLNS. The analysis in this section shows the strengths of tALNS when utilizing a tuned time-sensitivity factor.

4.6.3 Generalization

Table 9 shows the results of testing whether or not tALNS and LLNS can generalize to larger instance sizes. We can show that for our implementation, the operator preferred by LLNS still outperforms ALNS. tALNS also still performs much better than raw ALNS. Note that none of the algorithms is able to find solutions with GTO10 for instance 733, which features a very large amount of routes (at least 159 vehicles needed to serve all customers). The algorithms reach similar solution qualities in all instances, with LLNS doing so the fastest. To further evaluate the generalization capabilities of the algorithms, tests in different problem domains are needed. This is out of the scope of this study and subject of further research.

5 Conclusion

This work tests two enhancements to the Adaptive Large Neighborhood Search: Firstly, we show that adding time-sensitivity to the weight adjustment scheme can significantly speed up the ALNS, especially when many complex operators are present. We furthermore prove that including a tuned sensitivity factor, which is currently not common in the existing literature, is crucial for tALNS to perform well even in edge cases where the most efficient operators are slower than average. By outperforming classic ALNS even with small hand-picked operator portfolios by more than 30%, we show that the inclusion of tALNS is worth the added complexity (in our case, the timing operations slow the algorithm down by roughly 2%) and the additional meta-parameter. Further work is needed to prove that tALNS performs well outside our problem domain. Additionally, testing is needed to compare how tALNS behaves with pairwise operator selection, as the degree of destruction of a given destroy operator significantly influences the runtime of the following repair operator.

The second tested enhancement replaces the adaptive layer with a reinforcement learned operator selection policy, creating the Learned Large Neighborhood Search. We are able to learn a policy that consistently chooses the most efficient repair operator and significantly outperforms ALNS even after operator pre-selection and only using domain-agnostic features. Further testing is needed to determine whether or not a more efficient policy exists that utilizes more than one operator. Our training setup leaves room for improvement. Firstly, we spend around 99% of the considerable training time solving the vehicle routing problem. We estimate that we compute more than 50 million transitions

throughout this study. By saving these transitions for an offline training dataset, training itself and especially testing training configurations can be sped up massively. Secondly, a more substantial basis for operator selection can be formed by incorporating domain-specific features. This could be especially beneficial for more complex problem domains where specialized operators are needed to handle different aspects of the problem, for example, the vehicle assignment in CVRP with heterogeneous fleets.

References

Adulyasak, Y., Cordeau, J.-F., & Jans, R. (2014). Optimization-based Adaptive Large Neighborhood Search for the Production Routing Problem. *Transportation Science*, *48*(1), 20–45. http://www.jstor.org/stable/43666994

Ahuja, R. K., Ergun, Ö., Orlin, J. B., & Punnen, A. P. (2002). A Survey of Very Large-scale Neighborhood Search Techniques. *Discrete Applied Mathematics*, *123*(1-3), 75–102. https://doi.org/10.1016/S0166-218X(01)00338-9

Bellemare, M. G., Dabney, W., & Munos, R. (2017). A Distributional Perspective on Reinforcement Learning. http://arxiv.org/pdf/1707.06887v1

Ben-Ameur, W. (2004). Computing the Initial Temperature of Simulated Annealing. *Computational Optimization and Applications*, *29*(3), 369–385. https://doi.org/10.1023/B:COAP.0000044187.23143.bd

Bengio, Y. (2012). Practical Recommendations for Gradient-based Training of Deep Architectures. https://doi.org/10.48550/arXiv.1206.5533

Bengio, Y., Lodi, A., & Prouvost, A. (2021). Machine Learning for Combinatorial Optimization: A Methodological Tour D'horizon. *European Journal of Operational Research*, *290*(2), 405–421. https://doi.org/10.1016/j.ejor.2020.07.063

Bongiovanni, C., Kaspi, M., Cordeau, J.-F., & Geroliminis, N. (2022). A Machine Learning-driven Two-phase Metaheuristic for Autonomous Ridesharing Operations. *Transportation Research Part E: Logistics and Transportation Review*, *165*. https://doi.org/10.1016/j.tre.2022.102835

Boualamia, H., Metrane, A., Hafidi, I., & Mellouli, O. (2023). A New Adaptation Mechanism of the ALNS Algorithm Using Reinforcement Learning. In N. Aboutabit, M. Lazaar, & I. Hafidi (Eds.), *Advances in Machine Intelligence and Computer Science Applications* (pp. 3–14, Vol. 656). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-29313-9{\textunderscore}1

Canca, D., De-Los-Santos, A., Laporte, G., & Mesa, J. A. (2018). The Railway Network Design, Line Planning and Capacity Problem: an Adaptive Large Neighborhood Search Metaheuristic. In J. Żak, Y. Hadas, & R. Rossi (Eds.), *Advanced Concepts, Methodologies and Technologies for Transportation and Logistics* (pp. 198–219, Vol. 572). Springer International Publishing. https://doi.org/10.1007/978-3-319-57105-8{\textunderscore}9

Černý, V. (1985). Thermodynamical Approach to the Traveling Salesman Problem: an Efficient Simulation Algorithm. *Journal of Optimization Theory and Applications*, *45*(1), 41–51. https://doi.org/10.1007/BF00940812

Chen, X., & Tian, Y. (2018). Learning to Perform Local Rewriting for Combinatorial Optimization. https://doi.org/10.48550/arXiv.1810.00337

Da Costa, P., Rhuggenaath, J., Zhang, Y., Akcay, A., & Kaymak, U. (2021). Learning 2-Opt Heuristics for Routing Problems via Deep Reinforcement Learning. *SN Computer Science*, *2*(5). https://doi.org/10.1007/s42979-021-00779-2

Dantzig, G. B., & Ramser, J. H. (1959). The Truck Dispatching Problem. *Management Science*, *6*(1), 80–91. https://doi.org/10.1287/mnsc.6.1.80

Developers, T. (2023). Tensorflow. https://doi.org/10.5281/zenodo.4724125

Ferrari, S., & Stengel, R. F. (2005). Smooth Function Approximation Using Neural Networks. *IEEE Transactions on Neural Networks*, *16*(1), 24–38. https://doi.org/10.1109/TNN.2004.836233

Gao, L., Chen, M., Chen, Q., Luo, G., Zhu, N., & Liu, Z. (2020). Learn to Design the Heuristics for Vehicle Routing Problem. https://doi.org/10.48550/arXiv.2002.08539

Gendreau, M., & Potvin, J.-Y. (2010). *Handbook of Metaheuristics* (Vol. 146). Springer US. https://doi.org/10.1007/978-1-4419-1665-5

Godinho, M. T., Gouveia, L., & Magnanti, T. L. (2008). Combined Route Capacity and Route Length Models for Unit Demand Vehicle Routing Problems. *Discrete Optimization*, *5*(2), 350–372. https://doi.org/10.1016/j.disopt.2007.05.001

Gullhav, A. N., Cordeau, J.-F., Hvattum, L. M., & Nygreen, B. (2017). Adaptive Large Neighborhood Search Heuristics for Multi-tier Service Deployment Problems in Clouds. *European Journal of Operational Research*, *259*(3), 829–846. https://doi.org/10.1016/j.ejor.2016.11.003

Hochba, D. S. (1997). Approximation Algorithms for Np-hard Problems. *ACM SIGACT News*, *28*(2), 40–52. https://doi.org/10.1145/261342.571216

Hottung, A., & Tierney, K. (2022). Neural Large Neighborhood Search for Routing Problems. *Artificial Intelligence*, *313*, 103786. https://doi.org/10.1016/j.artint.2022.103786

Joe, W., & Lau, H. C. (2020). Deep Reinforcement Learning Approach to Solve Dynamic Vehicle Routing Problem with Stochastic Customers. *Proceedings of the International Conference on Automated Planning and Scheduling*, *30*, 394–402.

Johnn, S.-N., Darvariu, V.-A., Handl, J., & Kalcsics, J. (2023). Graph Reinforcement Learning for Operator Selection in the ALNS Metaheuristic. https://doi.org/10.48550/arXiv.2302.14678

Karimi-Mamaghan, M., Mohammadi, M., Meyer, P., Karimi-Mamaghan, A. M., & Talbi, E.-G. (2022). Machine Learning at the Service of Meta-heuristics for Solving Combinatorial Optimization Problems: A State-of-the-art. *European Journal of Operational Research*, *296*(2), 393–422. https://doi.org/10.1016/j.ejor.2021.04.032

Kerschke, P., Hoos, H. H., Neumann, F., & Trautmann, H. (2019). Automated Algorithm Selection: Survey and Perspectives. *Evolutionary computation*, *27*(1), 3–45. https://doi.org/10.1162/evco{\textunderscore}a{\textunderscore}00242

Kiefer, A., Hartl, R. F., & Schnell, A. (2017). Adaptive Large Neighborhood Search for the Curriculum-based Course Timetabling Problem. *Annals of Operations Research*, *252*(2), 255–282. https://doi.org/10.1007/s10479-016-2151-2

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. *Science (New York, N.Y.)*, *220*(4598), 671–680. https://doi.org/10.1126/science.220.4598.671

Kool, W., van Hoof, H., & Welling, M. (2018). Attention, Learn to Solve Routing Problems! In *International Conference on Learning Representations*. ICLR. https://doi.org/10.48550/arXiv.1803.08475

Kucukoglu, I., Dewil, R., & Cattrysse, D. (2021). The Electric Vehicle Routing Problem and Its Variations: A Literature Review. *Computers & Industrial Engineering*, *161*, 107650. https://doi.org/10.1016/j.cie.2021.107650

Laborie, P., & Godard, D. (2007). Self-adapting Large Neighborhood Search: Application to Single-mode Scheduling Problems. In *Proceedings MISTA-07, Paris, Vol. 8* (Vol. 8). Multidisciplinary International Conference on Scheduling.

Lima, I., Oliveira, D., & Queiroga, E. (2023). The Capacitated Vehicle Routing Problem Library (E. Queiroga, Ed.). Retrieved November 24, 2023, from http://vrp.galgos.inf.puc-rio.br/index.php/en/plotted-instances

Lu, H., Zhang, X., & Yang, S. (2020). A Learning-based Iterative Method for Solving Vehicle Routing Problems. In *International Conference on Learning Representation 2020*. Retrieved April 12, 2023, from https://openreview.net/forum?id=BJe1334YDH

Nazari, M., Oroojlooy, A., Snyder, L. V., & Takáč, M. (2018). Reinforcement Learning for Solving the Vehicle Routing Problem. https://doi.org/10.48550/arXiv.1802.04240

Oliva, D., Houssein, E. H., & Hinojosa, S. (2021). *Metaheuristics in Machine Learning: Theory and Applications*. Springer.

Pisinger, D., & Ropke, S. (2007). A General Heuristic for Vehicle Routing Problems. *Computers & Operations Research*, *34*(8), 2403–2435. https://doi.org/10.1016/j.cor.2005.09.012

Reijnen, R., Zhang, Y., Lau, H. C., & Bukhsh, Z. (2022). Operator Selection in Adaptive Large Neighborhood Search Using Deep Reinforcement Learning. https://doi.org/10.48550/arXiv.2211.00759

Ropke, S., & Pisinger, D. (2006). An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. *Transportation Science*, *40*(4), 455–472. https://doi.org/10.1287/trsc.1050.0135

Santini, A., Ropke, S., & Hvattum, L. M. (2018). A Comparison of Acceptance Criteria for the Adaptive Large Neighbourhood Search Metaheuristic. *Journal of Heuristics*, *24*(5), 783–815. https://doi.org/10.1007/s10732-018-9377-x

Shaw, P. (1998). Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In G. Goos, J. Hartmanis, J. van Leeuwen, M. Maher, & J.-F. Puget (Eds.), *Principles and Practice of Constraint Programming — CP98* (pp. 417–431, Vol. 1520). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-49481-2{\textunderscore}30

Song, J., Lanka, R., Yue, Y., & Dilkina, B. (2020). A General Large Neighborhood Search Framework for Solving Integer Linear Programs. https://doi.org/10.48550/arXiv.2004.00422

Sonnerat, N., Wang, P., Ktena, I., Bartunov, S., & Nair, V. (2021). Learning a Large Neighborhood Search Algorithm for Mixed Integer Programs. https://doi.org/10.48550/arXiv.2107.10201

Stathakis, D. (2009). How Many Hidden Layers and Nodes? *International Journal of Remote Sensing*, *30*(8), 2133–2147. https://doi.org/10.1080/01431160802549278

Thomas, C., & Schaus, P. (2018). Revisiting the Self-adaptive Large Neighborhood Search. In W.-J. van Hoeve (Ed.), *Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (pp. 557–566, Vol. 10848). Springer International Publishing. https://doi.org/10.1007/978-3-319-93031-2{\textunderscore}40

Turkeš, R., Sörensen, K., & Hvattum, L. M. (2021). Meta-analysis of Metaheuristics: Quantifying the Effect of Adaptiveness in Adaptive Large Neighborhood Search. *European Journal of Operational Research*, *292*(2), 423–442. https://doi.org/10.1016/j.ejor.2020.10.045

Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., & Subramanian, A. (2017). New Benchmark Instances for the Capacitated Vehicle Routing Problem. *European Journal of Operational Research*, *257*(3), 845–858. https://doi.org/10.1016/j.ejor.2016.08.012

Wang, P., Reinelt, G., & Tan, Y. (2012). Self-adaptive Large Neighborhood Search Algorithm for Parallel Machine Scheduling Problems. *Journal of Systems Engineering and Electronics*, *23*(2), 208–215. https://doi.org/10.1109/JSEE.2012.00027

Windras Mara, S. T., Norcahyo, R., Jodiawan, P., Lusiantoro, L., & Rifai, A. P. (2022). A Survey of Adaptive Large Neighborhood Search Algorithms and Applications. *Computers & Operations Research*, *146*, 105903. https://doi.org/10.1016/j.cor.2022.105903

Wu, Y., Yang, W., He, G., & Zhao, S. (2017). An Improved Adaptive Large Neighborhood Search Algorithm for the Heterogeneous Fixed Fleet Vehicle Routing Problem. *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 657–663. https://doi.org/10.1109/ICSESS.2017.8343000

Wu, Y., Song, W., Cao, Z., & Zhang, J. (2021). Learning Large Neighborhood Search Policy for Integer Programming. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, & J. Wortman Vaughan (Eds.), *Advances in Neural Information Processing Systems* (pp. 30075–30087, Vol. 34). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2021/file/fc9e62695def29ccdb9eb3fed5b4c8c8-Paper.pdf

Wu, Y., Song, W., Cao, Z., Zhang, J., & Lim, A. (2022). Learning Improvement Heuristics for Solving Routing Problems. *IEEE transactions on neural networks and learning systems*, *33*(9), 5057–5069. https://doi.org/10.1109/TNNLS.2021.3068828

Zhang, H., & Yang, W. (2021). An Enhanced Adaptive Large Neighborhood Search Algorithm for the Capacitated Vehicle Routing Problem. *2021 13th International Conference on Machine Learning and Computing*, 79–85. https://doi.org/10.1145/3457682.3457694

Zhou, J., Wu, Y., Cao, Z., Song, W., Zhang, J., & Chen, Z. (2023). Learning Large Neighborhood Search for Vehicle Routing in Airport Ground Handling. *IEEE Transactions on Knowledge and Data Engineering*, *35*(9), 9769–9782. https://doi.org/10.1109/TKDE.2023.3249799

List of Abbreviations

(¬)**ALNS** non-adaptive Large Neighborhood Search 2, 12, 14, 16, 17, 19–21

ALNS Adaptive Large Neighborhood Search 1–3, 6–11, 13–15, 17–22

ANN Artificial Neural Network 1, 2, 14

CO Combinatorial Optimization 1, 2, 6

CVRP Capacitated Vehicle Routing Problem 1, 2, 6–8, 23

DQL Deep-Q Learning 2, 7

DQN Deep-Q Network 13

DRL Deep Reinforcement Learning 6

eRTA expected runtime approximation 15, 17

eRTA-GTO5 expected runtime approximation value at 5% gap to optimum 13–19, 21

GTO gap to optimum 5, 13–17, 19, 21, 22

GTO5 5% gap to optimum 13, 14

LLNS Learned Large Neighborhood Search 2, 7, 8, 11, 13, 14, 17–22

LNS Large Neighborhood Search 2, 6, 8, 16, 18, 21

MDP Marcov Decision Process 6

ML Machine Learning 2, 6, 8

ns nano-second 15

OR Operations Research 1, 2, 6

PS pre-selected 18, 22

RL Reinforcement Learning 17, 21

RTA runtime approximation 11–15, 17–19, 21

SA Simulated Annealing 6, 8, 11, 13, 14, 17

SA-LNS Self-Adapting Large Neighborhood Search 3, 6

tALNS runtime-sensitive Adaptive Large Neighborhood Search 1, 2, 7, 8, 14, 17–22

TSP Traveling Salesman Problem 6

VRP Vehicle Routing Problem 2, 6, 8

Notation

α **update rate** for operator weights 4, 17

b **batch** of iterations 4, 8, 11

β **length of batch** in iterations 4, 8, 11, 17

c **cost** of solution $S \rightarrow c_S$ 10, 13

ct **cumulative runtimes** of operator o in batch $b \rightarrow ct_{o,b}$ 8

dod **degree of destruction** of destroy operator o, the percentage of nodes to remove out of the solution $\rightarrow dod_o$ 14, 17

ϵ **epsilon-greedy** factor deciding how often a policy should select a random action during training 14, 17, 18

gf **gap factor** for scaling the Gap to Optimum in the LLNS reward function 13, 15, 17

i **iteration** 8

ω **time sensitivity factor** used to scale the impact of runtime on tALNS weights 4, 7, 11, 17, 19–22

Ω^- **set** of destroy operators 7, 8

Ω^+ **set** of repair operators 7, 8, 11

p **probability** of operator being chosen in batch $b \rightarrow p_{o,b}$ 10, 11

ϕ **reward** for the acceptance check with outcome $\rho \rightarrow \phi_\rho$ 4, 8, 11, 13, 17

Φ **score** of operator o in iteration $i \rightarrow \Phi_{o,i}$ 8, 11

R **reward** gained for a given transition in Reinforcement Learning 13

ρ **result** of the acceptance check in iteration $i \rightarrow \rho_i$ 8, 11, 13, 17

S **active solution** 8, 10, 13

S^* **best solution** 8, 13

S^a **accepted solution** 8, 10, 13

sa_cr simulated annealing **cooling rate** to lower temperature 10, 11

sat simulated annealing **temperature** at iteration $i \rightarrow sat_i$ 8, 10

sa_ti simulated annealing **target iteration** for calculating cooling rate 10, 11

sa_tp simulated annealing **target percentage** for calculating cooling rate 10, 11

$S^{opt.}$ **optimal solution** 13

Θ **number of appearances** of destroy or repair operator o in batch $b \rightarrow \Theta_{o,b}^{-/+}$ 4, 8, 11

t **runtime** of destroy or repair operator o in iteration $i \rightarrow t_{o,i}^{-/+}$ 4, 8, 11

tf **time factor** for tALNS 4, 11

w **weight** of operator o in batch $b \rightarrow w_{o,b}$ 4, 8, 11