ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS
in
Combinatorial Decision Making & Optimization

MACHINE LEARNING FOR COMBINATORIAL OPTIMIZATION: THE CASE OF VEHICLE ROUTING

CANDIDATE
Diego Mazzieri

SUPERVISOR
Prof. Zeynep Kiziltan

Academic Year 2020/21

Session 3rd

Abstract

The Vehicle Routing Problem (VRP) is one of the most intensively studied combinatorial optimization problems in the Operations Research (OR) community. Its relevance is not only related to the various real-world applications it deals with, but to its inherent complexity being an NP-hard problem.

From its original formulation more than 60 years ago, numerous mathematical models and algorithms have been proposed to solve VRP. The most recent trend is to leverage Machine Learning (ML) in conjunction with these traditional approaches to enhance their performance.

In particular, this work investigates the use of ML-driven components as destroy or repair methods inside the Large Neighborhood Search (LNS) metaheuristic, trying to understand if, where, and when it is effective to apply them in the context of VRP.

For these purposes, we propose NeuRouting, an open-source hybridization framework aimed at facilitating the integration between ML and LNS.

Regarding the destroy phase, we adopt a Graph Neural Network (GNN) assisted heuristic, which we hybridize with a neural repair methodology taken from the literature. We investigate this integration both on its own and as part of an Adaptive Large Neighborhood Search (ALNS), performing an empirical study on instances of various sizes and against some traditional solvers.

Acknowledgements

This work marks the end of my 5-years long formative path as student inside the University of Bologna. First in Cesena, then in Bologna I have had the opportunity to know and collaborate with awesome people to whom I owe a lot both academically and personally. Among the others, I would like to thank Prof. Zeynep Kiziltan for having assisted me in this dissertation, but also for having supported my tutor position in the Combinatorial Decision Making course.

Regarding the feasibility of this research, due to hardware limitations and compatibility issues, I would not have been able to carry out any of the experiments without the help of the technicians of our university cluster, who have always been helpful and kind when I had problems.

I also want to thank my fellow students for their collaborative attitude and for having shared their passion about the world of AI beyond the strictly related aspects of university. I think being passionate of a topic is an invaluable perk in order to understand what you want to do professionally.

Nothing can be properly assessed without considering the context in which it occurs. This is why I would like to thank all the people who have been present daily in the last two years, starting from my family members Giuliano, Maria, and Sofia, to arrive at the friends I met across Jesi, Cesena, and Bologna.

Every time a cycle in my life ends I try to think of how I was when it started. In the case of this master degree I had lots of doubts and uncertainties regarding my choice. The fact it was of new activation, the doubt of not really liking a topic about which I nearly knew nothing about, and the added limitations related to the pandemic. Today, I can say with certainty I do not regret having undertaken this path, and I strongly believe the merits goes to all the people directly or indirectly mentioned here.

Contents

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ALNS	**A**daptive **L**arge **N**eighborhood **S**earch
ANN	**A**rtificial **N**eural **N**etwork
B&B	**B**ranch **and** **B**ound
CNN	**C**onvolutional **N**eural **N**etwork
CO	**C**ombinatorial **O**ptimization
CP	**C**onstraint **P**rogramming
CVRP	**C**apacitated **V**ehicle **R**outing **P**roblem
DL	**D**eep **L**earning
GAT	**G**raph **AT**tention Network
GCN	**G**raph **C**onvolutional **N**etwork
GNN	**G**raph **N**eural **N**etwork
LNS	**L**arge **N**eighborhood **S**earch
LP	**L**inear **P**rogramming
MILP	**M**ixed **I**nteger **L**inear **P**rogramming
ML	**M**achine **L**earning
MLP	**M**ulti-**L**ayer **P**erceptron
NLNS	**N**eural **L**arge **N**eighborhood **S**earch
NCO	**N**eural **C**ombinatorial **O**ptimization
OR	**O**perations **R**esearch
PN	**P**ointer **N**etwork
RL	**R**einforcement **L**earning
RNN	**R**ecurrent **N**eural **N**etwork
SAT	**SAT**isfiability Problem
SGD	**S**tochastic **G**radient **D**escent
TSP	**T**ravelling **S**alesman **P**roblem
VRP	**V**ehicle **R**outing **P**roblem

Chapter 1

Introduction

In order to provide a general idea of what the rest of the dissertation covers, this first introductory chapter outlines the context and the reasons behind our research. Section 1.1 presents the field of combinatorial optimization, focusing on the particular class of routing problems and giving some hints on why machine learning can be beneficial within this area, section 1.2 summarizes some relevant research questions not yet answered by the scientific community regarding these hybrid techniques, section 1.3 provides an overview of our contributions for answering them, and, finally, section 1.4 explains the organization of the following chapters.

1.1 Context

Operations Research (OR) is the area of mathematics concerning the development and application of analytical methods to improve decision-making. Born during the World War II as an initiative for military planning, nowadays, it forms the backbone of some of the most important industries, including but not limited to, transportation, telecommunications, logistics, scheduling, and supply chains [44].

OR problems are formulated using integer constrained optimization language (i.e., with integral or binary variables on which to perform decisions). While not all such problems are hard to solve (e.g., finding the shortest path between two locations), we concentrate on the subset of them belonging to the field of *Combinatorial Optimization* (CO). This kind of problems have the common characteristic of being NP-hard [32], which makes it impossible to solve them optimally at large scales as exhaustively searching for their solutions is beyond the limits of modern computers. The literature

about CO covers the rich set of techniques researchers have developed to tackle the issues related to this aspect. Take for instance the famous *Travelling Salesman Problem* (TSP), its state-of-the-art solver, *Concorde*[1], leverages over 50 years of research on linear programming, cutting plane algorithms, and branch-and-bound. These approaches have been collectively learned by the scientific community to address the inaccessible distribution of problem instances available.

Machine Learning (ML) focuses on performing tasks in domains for which no clear mathematical formulation emerges (e.g., images, text, voice, etc.). CO instances present this same characteristic and, therefore, are good candidates to be solved using this approach [5]. It is important to notice that the focus is not on substituting one with the other, rather, the merging between the two worlds aims at incorporating ML components in the CO algorithm to automatically perform decisions on a chosen distribution of instances.

The implicit knowledge extracted by ML algorithms is complementary to the explicit expertise extracted through CO research. From the CO point of view, ML can both go beyond the expert injected knowledge and replace heavy computations with a fast approximation. From the ML point of view, CO can decompose the problem into smaller, simpler, learning tasks. Despite being relatively new, there are already relevant examples in which this marriage has proven successful, the most noticeable ones being chip design [40] and protein folding [31].

Among the many CO problems available in the literature, we focus our attention on the *Vehicle Routing Problem* (VRP) [54], which concerns the design of the optimal set of routes involving a fleet of vehicles starting from a central depot, and required to serve a set of geographically scattered customers. The real-world applications of this problem are multiple, the most straightforward one being delivery services, but more generally including all the area of transportation.

Apart from its practical relevance, the interest in studying VRP from the scientific community lies in its inherent difficulty, being NP-hard. In order to overcome the limitations associated to this computational class, the majority of the state-of-the-art algorithms rely on handcrafted heuristics for making decisions that otherwise would be too expensive to compute [36]. Most recent approaches for facing VRP involve using ML

[1] `https://www.math.uwaterloo.ca/tsp/concorde/index.html`

to perform these decisions, either progressively constructing the final solution [42, 35], or starting from a feasible solution and iteratively improving it [12, 27].

Since we believe these latter methods are the most promising to be enhanced with ML in the case of VRP, we study their integration in the metaheuristic known as *Large Neighborhood Search* (LNS) [45], which is based on the idea of repeatedly relaxing a part of the actual solution using a *destroy* operation, and subsequently fixing it using a complementary *repair* operation.

1.2 Research Questions

Considering LNS applied to CO problems, research has focused either on destroy a solution or on repair a partial one using ML approaches for "learning to improve".

In the vehicle routing case, while for the repair heuristic we can take inspiration from the existing literature [27], with regard to the destroy methodology there is no promising candidate we can rely upon, given that, the work proposing it treat completely different problems and are strictly dependent on the usage of a *Mixed Integer Linear Programming* (MILP) solver [1, 51].

Which neural model can we employ for this task in the VRP case? Can hybridizing the neural destroy and repair outperform the already existing techniques? Even if not, when and where is it useful to employ neural-assisted operations among the different phases of the algorithm?

Despite being complementary to each other, the aforementioned research are not truly compatible; indeed, since each carries its own implementation of the LNS with the corresponding destroy and repair steps embedded inside, there exist an inherent difficulty in decoupling the different phases of the algorithm. The main issue related to this aspect lies in its poor extensibility: every time a new idea, either for destroying or for repairing the current solution, is developed, there is necessity to re-adapt the entire LNS metaheuristic. Is it possible to design it in a more modular way so that these limitations are overcome?

1.3 Contributions

Purpose of this research is to answer the questions presented in section 1.2:

- Regarding the destroy phase, we propose a *Graph Neural Network* (GNN) assisted approach, adapting the architecture in [34] for our LNS use case.
- We hybridize our neural-guided destroy with the neural-based repair architecture presented in [27], parameterizing the policy followed by the latter on the decisions taken by the former. Our interest does not lie solely in determining if a "completely neural" line of attack can outperform the already existing techniques for solving VRP, rather, we want to compare its effectiveness with respect to pairing ML-based heuristics with some *traditional* counterparts.
- We take into consideration the case of *Adaptive Neural Large Neighborhood Search* (ANLNS), in which multiple destroy/repair operation pairs, including neural enhanced ones, are jointly exploited to improve the quality of the current solution.
- We provide an open-source hybridization framework, *NeuRouting*[2], for comfortably experimenting different destroy/repair combinations in order to determine the most suitable ones in any scenario.

An empirical study is conducted solving a fixed set of instances under the same conditions, and then testing the generalization performance of our best approaches when dealing with instances, taken from the literature, more difficult than the ones used for training. We evaluate the aforementioned *neural* options with respect to: some stochastic methods as regards the destroy step, and to an exact procedure or a greedy heuristic for the repair phase.

1.4 Organization

The remainder of the dissertation is organized following the traditional structure of a scientific paper.

[2] `https://github.com/mazzio97/NeuRouting`

Chapter 2 provides all the theoretical background needed to understand the research done, starting from the notions of CO and ML up to the peculiar aspects of VRP.

Chapter 3 covers the existing literature about the integration of ML and CO, and the various techniques through which it can be achieved; with particular attention, but not limited to, routing problems.

The inner functioning of the neural methods we have included and a brief description of our LNS framework are detailed in chapter 4.

The various experiments led to determine the effectiveness of the proposed approaches are described in chapter 5, along with the adopted training methodology.

Finally, in chapter 6, we make our final considerations on the work done, outlining some future research directions which may be of interest to undertake.

Chapter 2

Background

This chapter aims to give the reader a primer knowledge about all the relevant concepts which we will make use of throughout the rest of the dissertation. Specifically, section 2.1 presents the field of combinatorial optimization and the most widely used techniques to deal with it, section 2.2 delves into the family of vehicle routing problems in particular, section 2.3 describes more in-depth how large neighborhood search can be applied for solving a routing problem, and section 2.4 provides a summarized description of some machine/reinforcement learning techniques which are needed to understand the methodologies adopted.

2.1 Combinatorial Optimization

Optimization problems, from a mathematical point of view, involve a set $S = x_1, x_2, ...$, either finite or infinite, in which each object $x \in S$ is linked to a cost computed using an objective function $f(x)$. To optimize $f(x)$ means to find the object $x^* \in S$ that returns the best value $f^*(x)$, which can be either the minimum or the maximum of f depending on the applications.

In *Combinatorial Optimization* (CO), the set S is finite. Though this may seem as a rather easy scenario, the problems to be solved generally fall under the NP-hard class of computation, thus they involve a huge number of objects, leading exhaustive search to be practically intractable.

2.1.1 Exact Methods

An exact method for solving a CO problem is a general purpose algorithm which guarantees to find the optimal solution. This kind of approaches, while being effective for some problems, usually present an exponential computational complexity, making them unsuitable to be used due to the huge amount a time that would be needed to provide optimal solutions.

CP-SAT

Without loss of generality, a CO problem can be formulated as a *constrained program* (CP). Constraints model natural or imposed restrictions of the problem, variables define the decisions to be made, while the objective function, generally a cost to be minimized, represents the measure of the quality of every feasible assignment of values to variables.

CP is an approach for solving discrete optimization problems which consists of two main components: propagation and search. Propagation is the process of communicating the domain reduction of a decision variable to all the constraints which involve that same variable. Search deals with the strategies to adopt when selecting a variable and when restricting the domain of its possible values.

Complementary to CP, we can use SAT solvers. The *satisfiability problem* (SAT) of propositional logic aims at determining if there exists an interpretation that satisfy a given boolean formula. It is one of the most well known problems in the computer science community, because it is the first which was proven to be NP-complete [13]. This characteristic enables many hard problems to be encoded as propositional formulas and efficiently solved using a SAT solver.

CP-SAT is a hybrid of CP and SAT that combines the expressivity of the former with the efficiency of the latter. It is based on the so-called *lazy clause generation* [52], which consists in recording the reasons which have led to a failure in previous propagations, to prevent the search repeating the same errors in future ones. The most well known CP-SAT solver is Google OR-Tools[1].

[1] `developers.google.com/optimization`

Mixed-Integer Linear Programming

If the objective and constraints are linear, the problem is called a linear programming (LP) problem. If, in addition, some variables are also restricted to only assume integer values, then the problem is a mixed-integer linear programming (MILP) problem.

The set of points that satisfy the constraints is the feasible region. Every point in that set (often referred to as a feasible solution) yields an upper bound on the objective value of the optimal solution.

With respect to complexity and solution methods, LP is a polynomial problem, well solved, in theory and in practice, through the simplex algorithm [17]. MILP, on the other hand, is an NP-hard problem. Indeed, it is easy to see that the complexity of MILP is associated with the integrality requirement on (some of) the variables, which makes the MILP feasible region non-convex. Dropping the integrality requirement defines a proper relaxation of MILP (i.e., an optimization problem whose feasible region contains the MILP feasible region), which happens to be a polynomially solvable LP.

Branch-and-bound (B&B) implements a divide-and-conquer type of algorithm representable by a search tree in which, at every node, an LP relaxation of the problem is efficiently computed. If the relaxation is infeasible, or if the solution of the relaxation is naturally (mixed-)integer (i.e., MILP feasible), the node does not need to be expanded. Otherwise, there exists at least one variable, among those supposed to be integer, taking a fractional value in the LP solution and that variable can be chosen for branching, i.e., by restricting its value in such a way that two child nodes are created. The two child nodes have disjoint feasible regions, none of which contains the solution of the previous LP relaxation.

Gurobi[2] and CPLEX[3] are two of the most well known MILP solvers available buying a proprietary license, while, among the free alternatives, SCIP[4] is the preferred choice.

[2] `https://www.gurobi.com/`
[3] `www.ibm.com/analytics/cplex-optimizer`
[4] `www.scipopt.org/`

2.1.2 Heuristics

It is not always possible or appropriate to apply exact solution methods due to basically two concurrent issues: the inner complexity of a CO problem (e.g., an NP-hard problem), and the time available to provide a solution, which may be limited. To this respect, it is important to clarify that the use of a heuristic method instead of an exact one must always be preceded by an attempt to formulate a model of the CO problem in the form of a MILP: this effort is useful to motivate the choice of the latter approach if the exact solution in a reasonable running time is not viable using the former.

It is also worth nothing that, while in some cases the availability of a provable optimal solution is necessary, in the vast majority of real scenarios a good approximate solution is enough, in particular for large size instances of a CO problem. In fact:

- For many parameters coming from a real application just estimates are available, which may be also subject to error, and it may be not worth waiting a long time for a solution whose value (or even feasibility) cannot be ensured.
- In a real-time system it is required that a "good" feasible solution is provided within a limited amount of time (e.g., few milliseconds).

These examples attest for the extended use of methods aiming at providing "solid" solutions and guarantee acceptable computing times, even if they cannot guarantee optimality: they are called heuristic methods (from greek *eur'ıskein* = to find).

In many CO cases, it is possible to devise some specific heuristic that exploits features of the problem itself and the human experience of who solves it in practice. In fact, very often, an optimization algorithm comes directly from coding the rules applied to "manually" solve the problem.

Constructive Heuristics

Constructive heuristics provide a solution by building it based only on input data and using a scheme that does not consider, or strongly limits, backtracking: they start from an empty solution and, iteratively, at each step, new elements are added to the solution according to a predefined expansion criterion, until a complete solution is defined. Among the many possible constructive heuristics, the most common are:

- **Greedy algorithms**: adopt a local expansion criterion, that is, the choice is the one which seems to be the best at that moment; at each iteration, the element to add to the current solution is the one that provides the best improvement to the objective function.

- **Exact solution embedding**: expansion criterion can be interpreted as an optimization (sub)-problem, which is easier than the original one.

- **Simplifying exact procedures**: while exact algorithms require an implicit enumeration (e.g., B&B) that gives rise to an exponential number of alternatives to be evaluated, we can select only a subset of them based on some criterion (e.g, stop after a certain depth, stop after a fixed time-limit, *beam search*).

Notice that all of these techniques are devised such that the overall final running time is short, which means the computational complexity is polynomial.

Improvement Heuristics

Given an optimization problem P, defined by an objective function f and a feasible region X. A neighborhood N is an application $N : s \rightarrow N(s)$ that associates, to each point $s \in X$ a subset $N(s) \subseteq X$.

The basic idea of the heuristic known as neighborhood search is the following: start from an initial (current) solution x and try to improve it by exploring a suitable neighborhood. If the neighborhood contains a solution x_0 better than x, then iterate the process considering x_0 as the new current solution.

The simplest version of neighborhood search is *local search*: the algorithm stops when the neighborhood of the current solution contains no improving candidates. However, this guarantees the current solution is a *local optimum* not a *global optimum*, and even if we let the process continue indefinitely it would not improve the already provided best result.

Metaheuristics, as the name suggests, are general algorithmic methods which are devised, independently of the specific CO problem, to guide the local search heuristic avoiding getting stuck in local optimums. There are a wide variety of metaheuristics [6], the most well known in the CO context being: *Simulated Annealing*, which

stochastically allows deteriorating solutions to be selected, *Tabu Search*, which prevents revisiting the same neighbors, and *Large Neighborhood Search*, which performs the investigation of promising candidates in a wider region of the solution space.

2.1.3 Large Neighborhood Search

As we already said, given a feasible solution to an instance of a problem, local search algorithms explore a neighborhood of that solution trying to improve it. This process can be performed using a *brute force* approach only if the size of the neighborhood is small, otherwise, we should employ techniques based on CP/MILP. In this latter case, we talk about large neighborhoods, and the usual methodology to search through them is by fixing a certain percentage of the available variables to their current value and solve the reduced version of the original problem consisting only of the remaining unfixed ones.

The advantage of having a large neighborhood is the possibility to explore a farther region of the solution space, and that makes it harder to get stuck in local minima. Indeed, the percentage of variables to fix is critical for the effectiveness of the algorithm, since if it is too large we risk spending too much time looking at the neighborhood of a single solution, while if it is too small we would not get any benefit with respect to using brute force.

Large Neighborhood Search (LNS) [45] is a metaheuristic firstly proposed by Shaw in 1998, which demonstrated particularly effective for solving vehicle routing problems [50]. While most neighborhood search algorithms explicitly define the neighborhood to explore, in LNS, it is implicitly determined by a *destroy* followed by a *repair* phase. The destroy procedure is mainly defined by a *degree of destruction*, representing the fraction of variables affected by the procedure, and typically contains an element of stochasticity such that different parts of the solution are destroyed at every invocation of the method. The repair procedure rebuilds the destroyed solution either using an exact or a heuristic algorithm.

The neighborhood $N(x)$ of a solution x is then defined as the set of solutions that can be reached by first applying the destroy method and then the repair method, as depicted in figure 2.1.

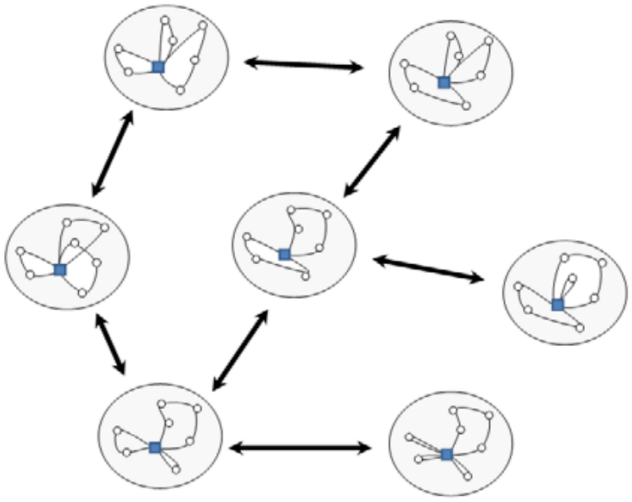


FIGURE 2.1: LNS conceptual representation.

The candidate solution for the next iteration is the best solution across the neighborhood, and its *acceptance* criteria can be designed in different ways apart from only allowing improving solutions as in the original formulation of Shaw [50]. For instance, Ropke & Pisinger [48] propose to use simulated annealing: a new solution is always accepted if of least cost than the current one or with probability $e^{-(c(x^t)-c(x))/T}$, where c is the cost function, x the current solution, x^t the candidate one, and T the temperature which is decreased gradually at each iteration to progressively refuse deteriorating solutions.

Two key concepts when designing a LNS algorithm are *diversification* and *intensification*. While the former deals with visiting unexplored regions to be sure that the search space is not confined to a reduced landscape, the latter explores more thoroughly promising regions in the hope to find better solutions. Usually, destroy methods aims at diversification while repair methods are intended for intensification, however, we can also perform intensification during destruction by removing variables considered critical according to some metric.

2.2 Vehicle Routing Problem

One of the most notorious applications of combinatorial optimization is vehicle routing (VRP) [54], in which the goal is to find the best routes for a fleet of vehicles visiting a

set of locations. It typically concerns the service of a delivery company: from one depot which has a set of vehicles who can move on a given road network to a set of customers, determine a set of routes, one for each vehicle, starting and ending at the depot, such that all customers' demands and operational constraints (e.g., capacity, time) are satisfied and the global transportation cost (e.g., fuel saving, delivery time, distance covered) is minimized.

The network can be described using a graph, where the edges are the roads and the vertices are the customers to visit plus the depot. In a real-world scenario, the cost associated to each edge can be easily computed using some shortest path algorithm. The travel time is the sum of the travel times of the arcs involved in each vehicle computed route.

2.2.1 Vehicle Routing Family

The term "vehicle routing" does not address a single CO problem in particular, rather, it covers an entire family of problems, each characterized by the introduction of new constraints or the relaxation of the ones defined in another formulation.

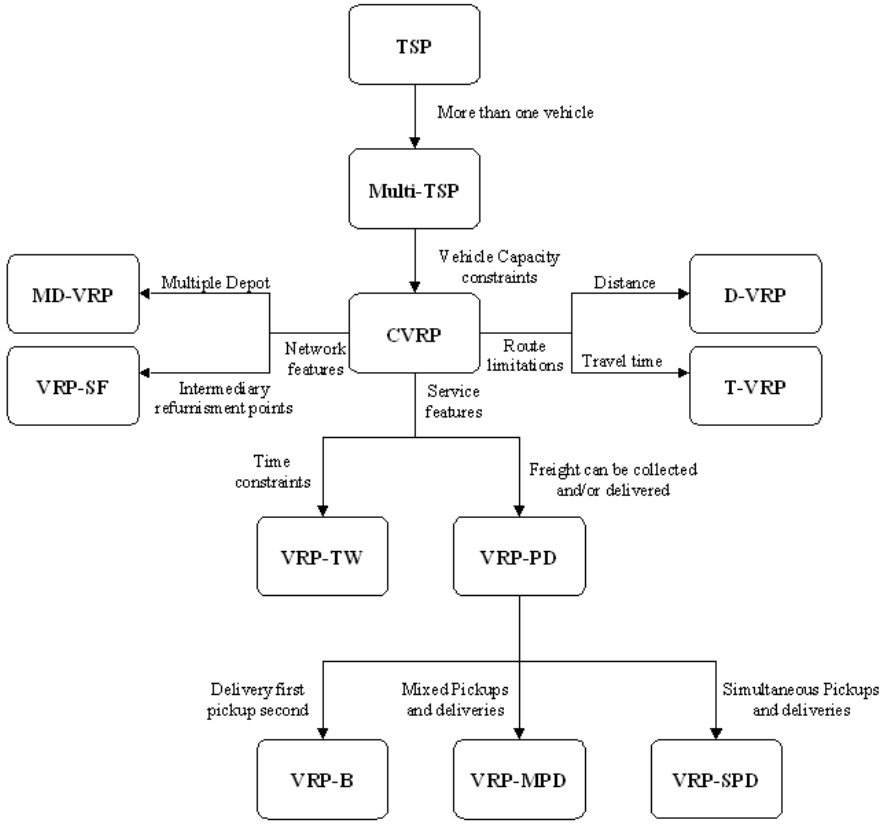
The most well known variants of VRP in literature are differentiated, among the others, for: the number of vehicles involved (the simplest case of one vehicle is referred to as TSP), the presence of capacity limits for the vehicles (CVRP), time-windows associated to each customer during which they must be visited (VRP-TW), possibility to collect items along the tour (VRP-PD).

A comprehensive map of the relation between the various vehicle routing problems is represented in figure 2.2.

2.2.2 Capacitated VRP

We focus our attention on the *Capacitated Vehicle Routing Problem* (CVRP) [16], which presents the following properties:

- $V = 0, 1, ..., n$ is the set of nodes, each with a different location.
- The node 0 represents the depot, while the nodes from 1 to n the customers with their respective demands q_i.



Source: Gonzalez-Feliu [22]

FIGURE 2.2: The "genealogical tree" of vehicle routing problems.

- The edge connecting two nodes i and j has a travel cost referred to as c_{ij}.
- All the p vehicles, where p is not given a priori, have the same maximum capacity Q.
- The objective is to serve all the customers, while never exceeding the capacity of the vehicles, traveling at the lowest possible cost.

Based on this description, we can give a mathematical formulation of our problem in the form of a MILP as follows:

$$\min \sum_{k=1}^{p} \sum_{i=0}^{n} \sum_{j=0}^{n} c_{ij} x_{ijk} \tag{2.1}$$

$$\text{s.t.} \sum_{i=0}^{n} x_{ijk} = \sum_{i=0}^{n} x_{jik} \qquad \forall j \in \{1,...,n\},\ k \in \{1,...,p\} \tag{2.2}$$

$$\sum_{k=1}^{p} \sum_{i=1}^{n} x_{ijk} = 1 \qquad \forall j \in \{2,...,n\} \tag{2.3}$$

$$\sum_{j=1}^{n} x_{1jk} = 1 \qquad \forall k \in \{1,...,p\} \tag{2.4}$$

$$\sum_{i=0}^{n} \sum_{j=1}^{n} q_j x_{ijk} \leq Q \qquad \forall k \in \{1,...,p\} \tag{2.5}$$

$$u_j - u_i \geq q_j - Q(1 - x_{ijk}) \qquad \forall i,j \in V \setminus \{0\}\ \ i \neq j \tag{2.6}$$

$$q_i \leq u_i \leq Q \qquad \forall i \in V \setminus \{0\} \tag{2.7}$$

$$x_{ijk} \in \{0,1\}, x_{iik} = 0 \qquad \forall k \in \{1,...,p\},\ \ i,j \in \{0,...,n\} \tag{2.8}$$

The binary variable x_{ijk} has a value of 1 if vehicle k drives from node i to node j, 0 otherwise. The expression (2.1) represents the objective function, (2.2) and (2.3) ensures that every node is entered and left only once, (2.4) and (2.5) verify that all vehicles start their tours at the depot and do not exceed the maximum capacity during it. Finally, (2.6) and (2.7) solve the subtours elimination problem, in which a vehicle route is composed by more than one connected component, using the Miller-Tucker-Zemlin formulation [18].

The best known solver entirely dedicated to VRP is LKH[5], which is based on the Lin-Kernighan-Helsgaun heuristic [25], originally proposed specifically for TSP, but then extended to other routing problems [24].

2.3 Large Neighborhood Search for CVRP

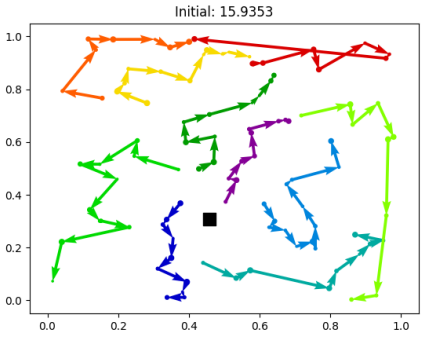
We have analyzed the characteristics of LNS in a general context, however, we are interested in particular to the CVRP, for which a formal definition has been given in

[5] `http://akira.ruc.dk/~keld/research/LKH-3/`

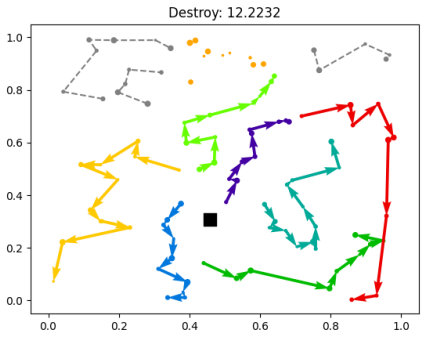
section 2.2. We can exploit this practical setting to further concretely explain how LNS works.

Take an instance of 100 customers and a feasible solution to that instance. Suppose our destroy operation is designed to destroy 10% of the solution, this translates in removing 10 customers by splitting the routes they belong to in two separate incomplete tours. Note that, despite the small percentage considered, there are $\binom{100}{10} = \frac{100!}{10!\times 90!} \approx 1.73 \times 10^{13}$ ways to select the nodes to remove.

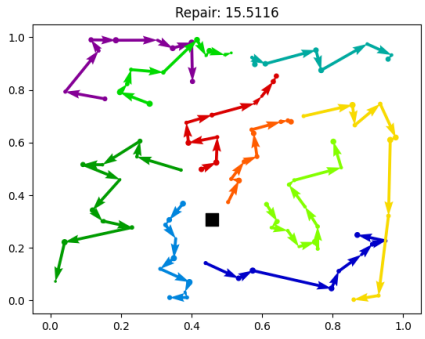
For each partial solution generated by the destroy method, the repair procedure takes a customer who has been removed and connects it to an incomplete tour or to another "isolated" customer. A hypothetical iteration of LNS in the discussed case is shown in figure 2.3.



(A) Feasible solution



(B) After destroy method



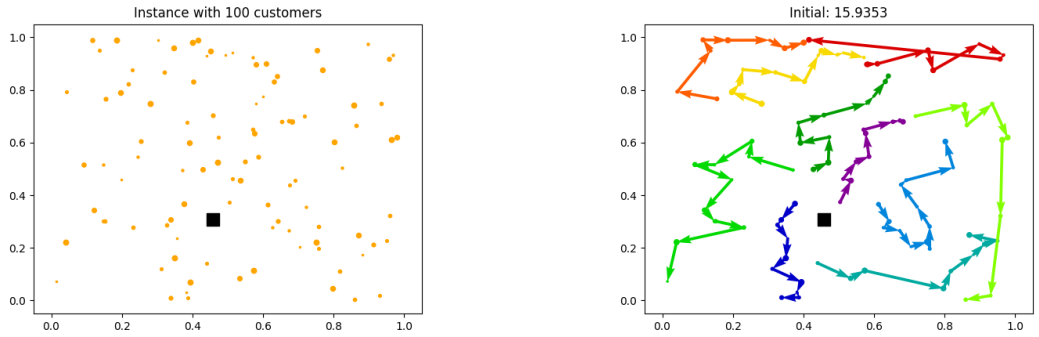
(C) After repair method

FIGURE 2.3: LNS iteration on a CVRP instance containing 100 customers. Different colors correspond to different complete routes. If gray the route is incomplete, which means it does not start and/or end at the depot. Orange nodes correspond to the removed customers.

2.3.1 Initial Solution

The first step of LNS consists in obtaining an initial solution for the CVRP instance taken into consideration. One of the most common approaches consists in constructing it following a greedy selection heuristic: starting from the depot, visit every time the nearest customer from the current location as long as the vehicle capacity is sufficient to satisfy the demands of the served customers, otherwise go back to the depot. Repeat the same policy until there are no missing customers to go to.

A more refined solution can be obtained using one of the constructive approaches discussed in section 3.1. However, these are often computationally heavier algorithms than a simple heuristic, which take away time to the searching procedure and do not provide any guarantees of obtaining a better result just because the starting solution is more promising.



(A) Generated instance (B) Greedy solution

FIGURE 2.4: An instance with 100 customers and the corresponding greedy solution routes.

2.3.2 Destroy Methods

The destroy method has a crucial role inside the LNS framework. If only a small part of the solution is destroyed, then the benefits of having a large neighborhood are lost. Conversely, if a large part is removed, dependent on how the partial solution is repaired, we could fall in time-consuming iterations or poor quality solutions.

Moreover, the destroy procedure must also be designed in such a way that the entire search space can be reached, therefore, it should make it possible to destroy every part of the solution and not focus only on destroying a little fraction of it.

The simplest method of selection of customers is obviously at *random*. Recalling the previous example instance of 100 customers and a degree of destruction of 10%, this algorithm selects stochastically 10 customers without replacement and removes their incoming and outcoming edge from the current solution.

Alternatively, we can exploit the "relation" between the variables to determine which customers remove. This is the case of *point-based* destruction, which, once sampled the coordinates of a point in the map, takes the 10 customers closest to that position.

Maintaining the same stochastic algorithm of the previous method, the *tour-based* destroy determines the nearest route, deletes all the customers belonging to it and repeats the procedure until at least the desired number of nodes has been picked.

2.3.3 Repair Methods

Repair methods are often based on some approximated or exact algorithms for the given problem. While the former are usually domain specific greedy-based heuristics, the latter are typically performed using an optimization suite and can be relaxed to reduce time resources at the expense of solution quality.

A typical *greedy* repair algorithm for CVRP can be designed selecting the customers which are not in any "complete" route (i.e. starting and ending at the depot) nor in the middle of a "partial" route, and connecting each of them to the nearest node presenting these same properties without violating the capacity constraints of the vehicle. In order to introduce some sort of diversification, the sequence in which the nodes are processed could be produced stochastically.

On the other hand, a *MILP* reparation is an exact method which relies on an optimization solver (e.g., SCIP) to solve the integer programming formulation of the subproblem where all but the incoming and outcoming edges of the removed nodes are already fixed.

2.4 Machine Learning

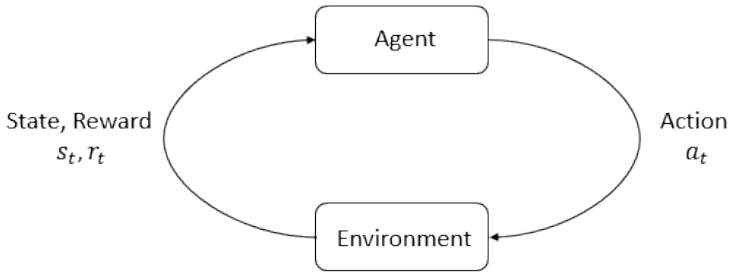
Machine Learning (ML) is the area of Artificial Intelligence which involves algorithms that are not explicitly programmed to solve a task but rather can improve their behavior

automatically. Indeed, models are fed with a large quantity of data samples and, via a learning procedure, they try to learn the unknown statistical distribution of the phenomenon that data belongs to. This training procedure is aimed at minimizing a loss function, which assumes different forms depending on the learning method, i.e., *supervised* – the ground truths are known –, *unsupervised* – the ground truths are not known –, and *reinforcement* learning – there are no ground truths, but rather reward mechanisms.

2.4.1 Reinforcement Learning

Reinforcement Learning (RL) [53] is concerned with how autonomous and adaptive agents behave and take actions. Differently from supervised learning, in RL the learning procedure involves some kind of reward mechanism, and it is aimed at maximizing the future cumulative reward, similarly to how biological learning works.

RL is a trial and error process where an *agent* performs *actions* in an environment. At each step the agent has a *state* and transitions from it to a new one receiving a *reward*, as represented in figure 2.5. The purpose is to learn the optimal *policy* (i.e., a mapping between a state and an action) to follow in order to maximize that reward over time (i.e., the so-called *value*).



Source: `https://spinningup.openai.com/`

FIGURE 2.5: The high-level worflow of a reinforcement learning algorithm.

Based on the actual procedure through which the learning occurs, we can subdivide RL into two different categories: *model-based* methods, which focus on the environment knowing its transition functions (e.g., board games), and *model-free* methods, which do not take the environment into consideration and solely utilize the experience collected by the agent.

The main upside of model-based approaches is that they allow the agent to plan by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between its options. The main downside is that a ground-truth model of the environment is usually not available or, even if it is, presents a too large space to be usefully explored.

That is the reason why model-free algorithms are more popular and have been more extensively studied. In particular, we can distinguish between *value-based* and *policy-based* methods.

Value-based Methods

Value-based methods, whose major exponent is DQN [41], are based on the concept of action-value function $Q_\pi(s,a)$, which is a measure of the expected reward if we start in state s, take an arbitrary action a, and then act according to the policy π.

Methods in this family learn an approximator $Q_\theta(s,a)$ for the optimal action-value function, $Q^*(s,a)$. This optimization is almost always performed *off-policy*, which means that each update can use data collected at any point during training. This property yields the advantage of being substantially more sample efficient, because data can be reused more effectively than when adopting policy-based techniques.

The corresponding policy π_θ can be obtained always selecting the best action in the current state according to Q_θ:

$$a(s) = \arg\max_a Q_\theta(s,a)$$

Policy-based Methods

Methods in this category – REINFORCE [60] to name one – explicitly represent the policy as a function $\pi_\theta(a|s)$, whose output is a probability distribution over all actions. They optimize the parameters θ directly by gradient ascent on the performance objective $J(\pi_\theta)$. This optimization is almost always performed *on-policy*, which means that each update only uses data collected while acting according to the most recent version of the policy.

The primary strength of policy-based with respect to value-based methods is that they are more stable and reliable, because they directly optimize the function we want to obtain, not one from which we can derive it.

Actor-Critic Methods

The aim of *actor-critic* methods is to combine the advantages from both value-based and policy-based approaches.

The principal idea is to split the model in two parts. The actor takes as input the state and outputs the best action, essentially controlling the agent behavior by learning the optimal policy (i.e., policy-based). The critic, on the other hand, evaluates the action by computing the Q value function (i.e., value based). This way, the critic provides the measure of how good the action taken by the actor has been, which allows to appropriately adjust the learnable parameters for the next train step.

It is like if the two models participate in a game where they both get better in their own role as the time passes. The result is that the overall architecture will learn to play the game more efficiently than the two methods separately.

2.4.2 Supervised Learning

In supervised learning the ground truths (i.e., labels) are known, therefore they can be explicitly used in the loss function, which assumes the form: $\mathscr{L}(y - \mathscr{M}(x, \theta))$, where y are the ground truths, x the input samples, $\mathscr{M}$ the ML model, and θ the vector of its learnable parameters.

The choice of $\mathscr{L}$ is critical and strictly depends on the kind of problem we are dealing with. In particular, if in presence of a *regression* problem, where we need to predict a real-valued quantity, *Mean Squared Error* is the most widely used loss function and consists in averaging the squared differences between the predicted and ground-truths values of the samples in the training set. Conversely, if facing a *classification* problem, where, for each sample, we need to select a label among a set of available ones (e.g., determine what an image is about between different categories), the default loss choice goes to *Cross-Entropy*. It calculates a score that summarizes the average

difference between the actual and predicted probability distributions for all categories in the problem.

Loss functions are a wide and complex topic in ML, comprising a lot of variations and adaptations of the aforementioned alternatives. However, the optimization algorithm used to minimize them is almost always *stochastic gradient descent* (SGD) or one of its variants (e.g., RMSProp, Adam, etc), which consists in iteratively updating the parameters θ by a small amount, controlled by the so-called *learning rate* α, going in the opposite direction of the gradient of the function to minimize:

$$\theta_{i+1} = \theta_i - \alpha \times \nabla_\theta \mathscr{L}(y - \mathscr{M}(x, \theta)))$$

The term "stochastic" comes from the limited number of data samples (i.e., usually referred to as a *batch*) used to compute the loss before applying a gradient update. Theoretically, this number should be equal to the number of items in the dataset at disposal, however, it becomes computationally too expensive to consider them all at each iteration.

Supervised learning is the most used approach for data-driven applications, however, ground truths are not always provided or easy to access. For instance, when trying to build an ML model that mimics a CO solver, learning in such a way is usually not recommended, because the performance of the model is tied to the quality of the supervised labels, and getting high-quality labeled data is often expensive if not even infeasible.

Up to now we have referred to the term "model" considering it as a *black-box*. An ML model is an algorithm able to exploit the hidden patterns present in the data it is trained with. It can assume various forms and should be selected depending on its suitability for the task to be solved. Among the most notorious supervised models we can mention: *Linear Regression*, *Logistic Regression*, *Support Vector Machines*, *Decision Trees*, and *Artificial Neural Networks*. This last type of models deserve a special mention because of their peculiar property of being universal function approximators [26], which makes them the most valuable choice in any complex scenario.

2.4.3 Deep Learning

Deep Learning (DL) [23] is a subfield of ML concerned with algorithms inspired by *Artificial Neural Networks* (ANN), computational models consisting of parametric composable functions in high dimensional spaces, which mimic the structure and the operation of the brain.

The most astonishing aspect of this kind of models is their ability to perform automatic feature extraction from raw data, also called *representation learning*, which let them approximate even complex functions whose inputs and outputs are far apart (e.g., an image with a text describing it).

The simplest ANN architecture is called *Multi-Layer Perceptron* (MLP) and consists in taking the input data in a vectorized form and progressively pass it through each layer constituting the model until obtaining the desired output. When passing through a layer, an affine transformation is applied on the vector, followed by a non-linear scalar function, known as the *activation function*, applied element-wise. The term "deep" comes exactly from the large number of layers, and consequently of parameters, which are usually employed.

Not all types of data present the same structure. For instance, text and time series are a typical example of sequential data, because the order of the elements matters. Conversely, images are the perfect example of spatial data, because a represented object remains so irrespective of its position. Depending on the practical application, an ANN might be more suitable than another, therefore research has been focused on developing different architectures depending on the domain of interest. In particular, tasks involving spatial data are usually faced using *Convolutional Neural Networks* (CNN), while task involving sequential data with *Recurrent Neural Networks* (RNN). There are plenty DL models in literature, each designed for a peculiar task. However, the most important aspect is that, regardless of their architecture, the usage and the way they learn is very similar to plain MLP networks.

Attention Mechanisms

Attention mechanisms are one of the greatest breakthrough in the DL community over the last decade. They represent a processing technique for neural networks that allows

the model to focus on specific parts of a complex input, similarly to how the visual attention mechanism of humans does.

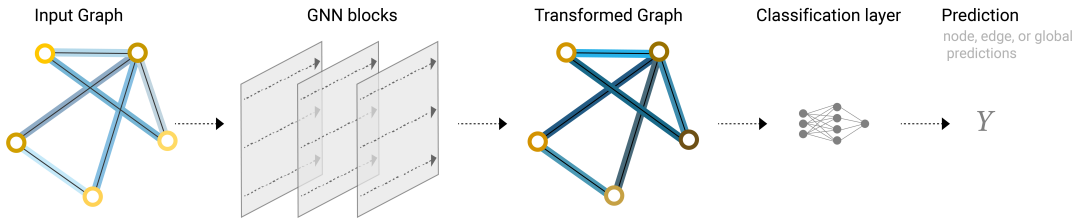
Originally emerged as an improvement of the *sequence-to-sequence* architectures [3], typically employed for machine translation tasks, their importance is related the central role they have in the *Transformer* architecture by Vaswani *et al.* [56].

The key idea is to compute a score describing the relationship between each pair of elements belonging to distinct sequences or even the same one (i.e., self-attention). This aspect has applicability in a large amount of domains beyond language translation, ranging from object detection [10] to image generation [47].

Graph Neural Networks

The generalization of an attention mechanism for the graph domain is a *Graph Neural Network* (GNN), in which, instead of all attending each other (forming a fully-connected graph), nodes interact only if they are connected by an edge [28].

While there are multiple variants of the original formulation [62], the most popular ones being *Graph Convolutional Networks* (GCN) [33] and *Graph Attention Networks* (GAT) [57], the key idea of all GNNs is to progressively transform the embeddings of the graph attributes (i.e., nodes, edges, global-context) without changing the connectivity of the input graph, and then using them to perform predictions, as visually represented in figure 2.6. A good introduction to GNNs is the article of Sanchez-Lengeling *et al.* [49].



Source: `https://distill.pub/2021/gnn-intro/`

FIGURE 2.6: The general end-to-end prediction task steps using a GNN model.

Chapter 3

Neural Combinatorial Optimization

Progress made in the last few years in the field of DL, made it possible to expand the application of such a technique to a wider set of both industrial and research areas. CO is one of them; indeed, the plethora of scientific papers published on these themes in the last few years, shows how the combination of the two approaches, commonly identified with the term *Neural Combinatorial Optimization* (NCO), can be beneficial under many aspects [5, 39, 58, 37].

Throughout this chapter, we analyze the different studies related to NCO led during the last five years, paying particular attention to those concerning routing problems. Each section covers a different methodology of applying DL to CO; specifically, in section 3.1 the model is delegated to directly construct the solution (which may be then refined), in section 3.2 the model assists an existing solver in the decisions it takes or configure its parameters depending on the instance to solve, and in section 3.3 a heuristic approach is enhanced using neural-based operations.

3.1 Learn to Construct

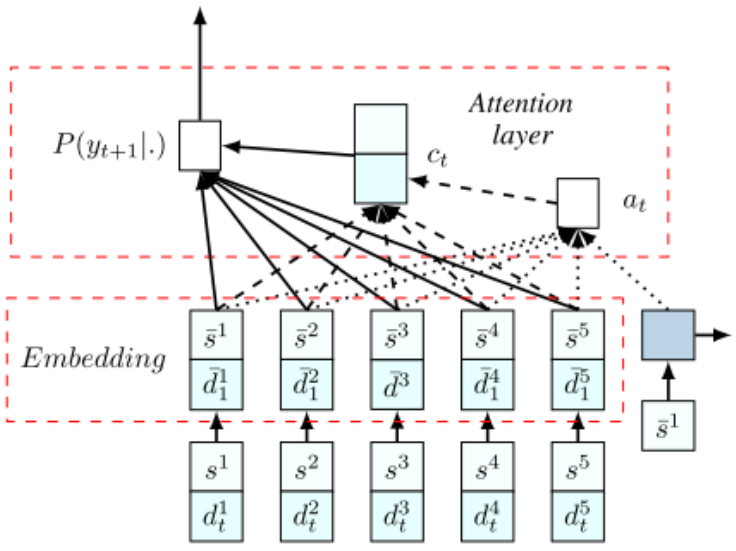
One of the most investigated ideas is to train the DL model to output solutions directly from the input instance. This kind of approach is commonly known as *end-to-end*, since the model acts as a black box and independently perform the learned policy constructing the result node after node.

One of the first attempts to use DL in a routing problem is the one of Vinyals *et al.* [59] taking into consideration TSP, the variant of VRP which provides only one vehicle.

The model used is a *Pointer Network* (PN), a variation of an RNN with attention mechanisms which outputs a permutation of the input sequence, and is trained in a supervised manner using as labels the solutions generated by a supervisor solver. The architecture follows the *encoder-decoder* pattern, and the decoding process is autoregressive: in order to predict the next node to take, the model considers as input the actions taken from the previous time steps.

Based on this pioneer work, Bello *et al.* [4] propose to use reinforcement learning instead of supervised learning to train the same model. This choice is motivated by the difficulty to obtain good labels (i.e., solutions) when the considered instances are large. The reward function they use in their actor-critic algorithm is the negative tour length of the produced solution.

Nazari *et al.* [42] keep the same training procedure but enhance the model to address also CVRP. Their contribution consists in replacing the original encoder, sensible to the nodes input order, with a permutation invariant embedding based on the position and on the demands of the customers. The decoding process is still based on a PN coupled with attention mechanisms. The architecture is illustrated in figure 3.1.



Source: Nazari *et al.* [42]

FIGURE 3.1: Nazari *et. al.* end-to-end model inner architecture.

Using actor-critic reinforcement learning as in [4, 42], but substituting the original encoder architecture with a *Transformer* network, Deudon *et. at.* [19] construct a solution which is then refined using the 2-Opt heuristic [14], claiming better results than

[4].

As in the architecture used in [19], also Kool *et al.* [35] opt for a *Transformer* network trained using reinforcement learning. However, in this latter case, they decide to: sample solutions according to the learned policy probabilities instead of performing a heuristic refinement, cover together with TSP other routing problems including CVRP, and interpret the attention mechanism as a weighted message passing algorithm providing useful information during the decoding step.

GNNs are among the most adopted models for solving CO problems, an overview of the existing approaches employing them is presented by Cappart *et al.* [9]. The first attempt to use a GNN to solve TSP is the one proposed by Dai *et al.* [15]. The model, trained using DQN, takes a graph and a partial solution as input, and outputs a state-value function Q from which greedily estimate the next node in the tour.

Nowak *et al.* [43] explore supervised learning of a GNN to solve small TSP instances (i.e., $n = 20$ nodes) from scratch, but obtaining slightly worse results than [4].

On top of this experimental work, Joshi *et al.* [30] propose a non-autoregressive approach: a GCN is trained, using supervision, to output the probabilities each edge belongs to the optimal TSP tour. Subsequently, a *beam search* is guided according to the generated "heatmap" to produce the solution. Following the same pipeline, Kool *et al.* [34] modify the GCN model in order to adapt it for CVRP.

3.2 Learn to Configure

Instead of directly tackling the problem, ML can be applied to provide additional pieces of information to a traditional CO solver. The main advantage of these methods, unlike pure ML ones, is that they can also prove the optimality and the feasibility of the produced solutions. However, their results when dealing with NP-hard discrete optimization problems, like vehicle routing ones, are usually of less quality. Moreover, this hybridization is challenging to build, as standard CP frameworks do not natively include machine learning mechanisms, leading to multiple sources of inefficiencies.

MIPLearn [2] is a supervised framework compatible with *Gurobi* and *CPLEX* commercial MILP solvers. Given some training instances, its purpose is to determine the

configuration parameters of the optimizer which are most likely to give good results when dealing with the same data distribution.

Instead of presetting the solver, Gasse *et al.* [21] focus on learning the branching decisions using a GNN on the variable-constraint bipartite representation of the MILP.

Generalizing the approaches proposed in the two aforementioned works, Prouvost *et al.* [46] introduce *Ecole*, a library for defining, through reinforcement learning, all the inner aspects of the free optimizer *SCIP*, which, in this way, acts as a controllable algorithm.

Cappart *et al.* [8] focus on injecting ML decisions inside a CP solver, namely *Gecode*, instead of a MILP solver. The main issue of using this approach is extensibility: the solver must be internally modified to accept neural assistance, and that implies it must also be open-source.

With the aim of overcoming these limitations, Chalumeau *et al.* [11] propose *SeaPearl*, a new CP solver written entirely in *Julia*[1], which natively supports machine learning routines in order to learn branching decisions using reinforcement learning. While representing a flexible framework that can facilitate future research in the hybridization of constraint programming and machine learning, *SeaPearl* is not yet competitive with industrial solvers.

3.3 Learn to Improve

The methods discussed in section 3.1 focus on learning heuristics that incrementally build a complete solution. Despite being comparatively fast, their results are usually worse with respect to the ones of traditional solvers. In order to narrow this gap, additional procedures, such as sampling or beam search, are usually exploited. However, since they rely on the same policy used for construction, exploration capabilities are intrinsically limited.

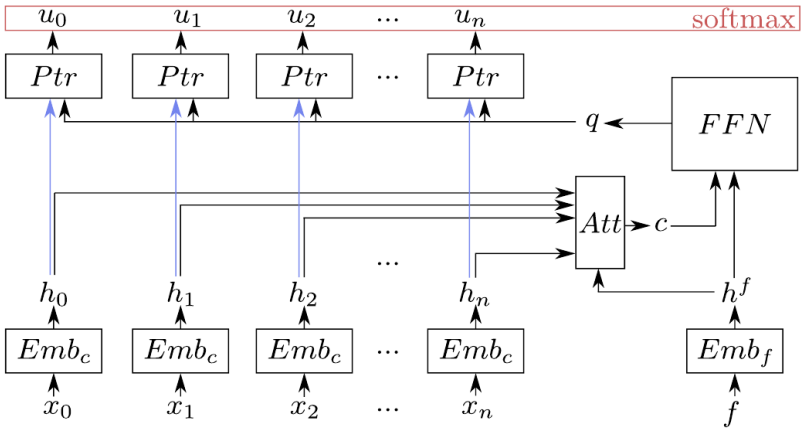
Rather than learning *construction* heuristics, a parallel trend focuses on directly learn *improvement* heuristics, which enhance an initial solution by iteratively performing neighborhood search based on certain ML operators, towards the direction of improving solution quality.

[1] `https://julialang.org/`

These operators, in the case of Lu *et al.* [38], are multiple man-made heuristics, which are selected by a machine learning model depending on the current solving state. In particular, along with improvement operators (e.g., 2-Opt), also perturbation operators, useful to escape local minima, are available.

Differently, Chen & Tian [12] and, subsequently, Wu *et al.* [61] propose to directly let the ML model perform the improvement step on the current solution. Specifically, a *region-picking* policy selects the fragment of solution to be improved, while a *rule-picking* policy executes the rewriting operation applicable to that region. Both works use an actor-critic algorithm for training, but the former relies on a *Long Short Term Memory* (LSTM) network, while the latter on a *Transformer* network.

The same attitude of learning the heuristics to apply instead of employing human-designed algorithms, is followed by Hottung & Tierney [27]. The innovative aspect of this work is that this kind of approach is integrated in a *Large Neighborhood Search* (LNS) setting as the "repair" mechanism, and applied specifically to CVRP. The actual model architecture which performs the operation is shown in figure 3.2.



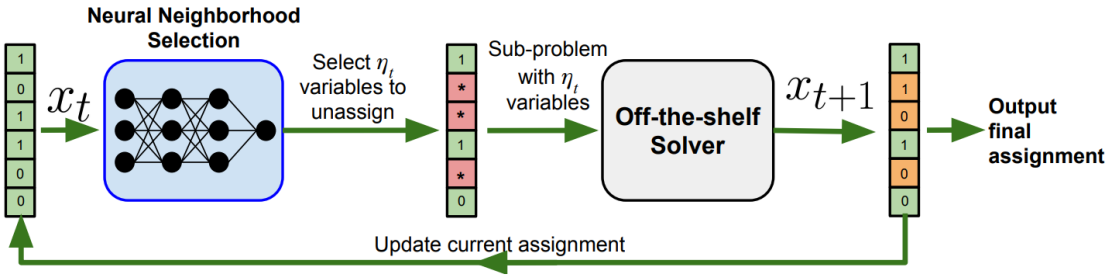
Source: Hottung & Tierney [27]

FIGURE 3.2: Hottung & Tierney neural repair model inner architecture.

For each partial tour $x_i \in X_t$ generated after a stochastic solution destroy, an embedding h_i is computed using the encoder Emb_c. The same is done for a randomly selected tour end f_t, which is transformed in h^t by a different encoder Emb_f. The computed embeddings pass through an attention layer Att, whose output is a context vector c describing the relevance of the inputs with respect to the tour end. Subsequently, the vector c is concatenated with h^t and given to a two-layer feed-forward network which

produces the vector q. Finally, based on $h_0 ... h_n$ and q, the logits $q_0 ... q_n$ are computed and the *softmax* operation is performed to obtain a probability distribution over all the actions, each corresponding to an x_i.

Instead of learning the "repair" policy, Addanki *et al.* [1] and Sonnerat *et al.* [51] propose to learn the "destroy" policy, and then to use an "off-the-shelf" MILP solver, in their case *SCIP*, to optimally reconstruct the resulting sub-problem, where all the variables except the removed ones are fixed, as it is explained in figure 3.3.



Source: Sonnerat *et al.* [51]

FIGURE 3.3: Starting from a feasible assignment of the variables of a MILP problem, at each iteration, a neural selection policy destroys η_t of them and an "off-the-shelf" solver repairs the resulting sub-problem.

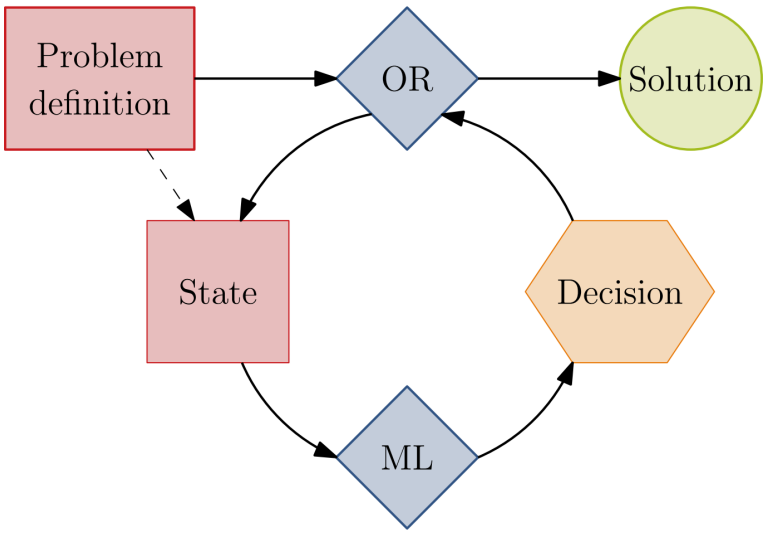
The neighborhood selection is operated by a GCN on the bipartite graph representation of the MILP, where the nodes are the variables plus the constraints defining the problem, while the edges only exist between a constraint node and the variables' nodes it involves. Their approach is evaluated on four different CO problems not including any VRP variant.

Chapter 4

Neural LNS for CVRP

Among the techniques for learning how to improve a solution, discussed in section 3.3, we focus our attention on the LNS iterative procedure applied in the context of CVRP. This problem, like many other CO ones, is highly structured and high-dimensional, making ANN the most promising candidates to parameterize the heuristic policies in charge of performing decisions either to destroy or to repair a solution.

The approach followed in this work falls into the third category presented by Bengio *et. at.* [5], namely the use of machine learning alongside optimization algorithms. It consists in repeatedly querying the same ML model to make decisions based on the current state, which may or may not include the problem definition. The workflow of this paradigm is depicted in figure 4.1.



Source: Bengio *et al.* [5]

FIGURE 4.1: Machine learning alongside optimization algorithms workflow.

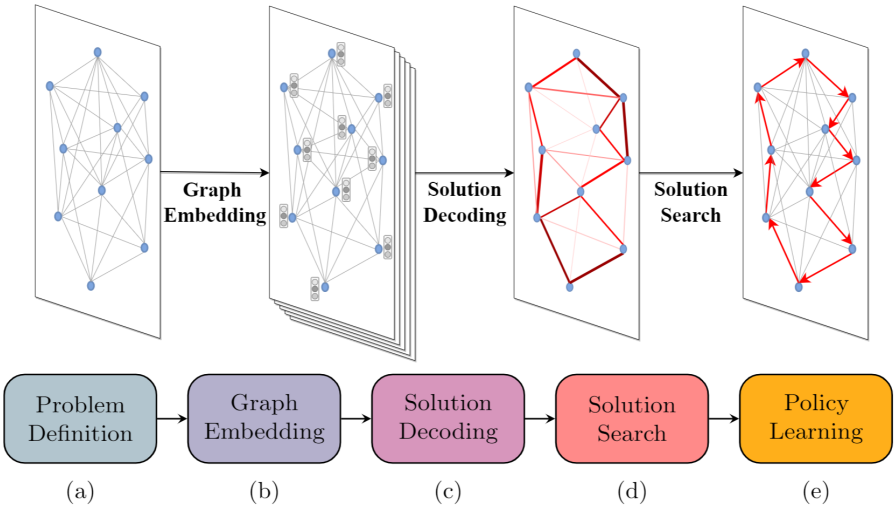
The structure of the chapter is designed as follows: section 4.1 presents the neural components and the way they are used inside our LNS framework, while section 4.2 briefly describes the *NeuRouting* architecture and available tools.

4.1 A Hybrid Approach

Neural Large Neighborhood Search (NLNS) is an extension of the original LNS meta-heuristic (see 2.1.3 for further details), where destroy and/or repair methods are guided using DL-based algorithms. The purpose is to exploit the generalization capabilities of DL with the search efficiency of LNS in order to obtain the best from both worlds.

4.1.1 Neural Destroy

GNNs represent one of the most promising research directions in NCO for routing problems, since they naturally operate on their intrinsic structure. The pipeline which is usually followed resembles the one of figure 4.2:



Source: `https://github.com/chaitjo/learning-tsp`

FIGURE 4.2: Neural Combinatorial Optimization using GNNs pipeline.

(a) The combinatorial problem is formulated via a graph.

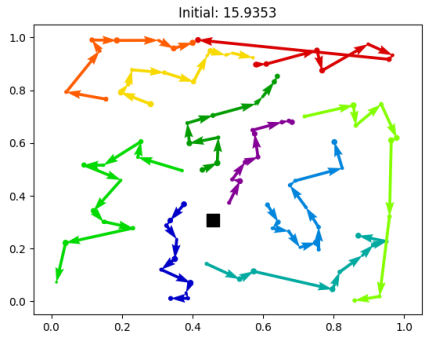
(b) Embeddings for each graph node are obtained using an encoder.

(c) Probabilities are assigned to each node for belonging to the solution set.

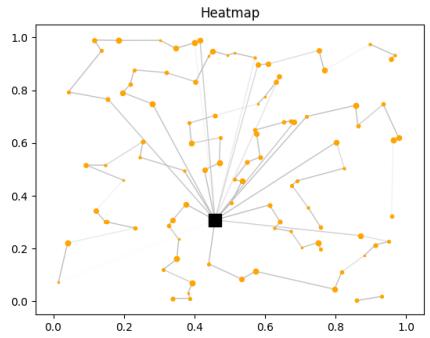
(d) The predicted probabilities are converted into discrete decisions.

(e) The entire model is trained via imitating an optimal solver (i.e., supervised learning) or through minimizing a cost function (i.e., reinforcement learning).

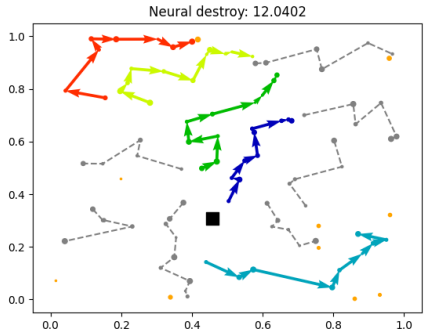
We have designed our destroy method based on the work of Kool *et. at.* [34], where a *Residual Gated GCN* [7] is used to predict a heatmap of the promising edges of an instance. In particular, we apply the same network masking all the edges but the ones in the current solution, obtaining a likelihood estimation of each of them as depicted in figure 4.3. The generated heatmap guides the selection of the edges to remove, since those presenting a low probability value are likely to improve the solution if replaced.



(A) Current solution



(B) Generated heatmap



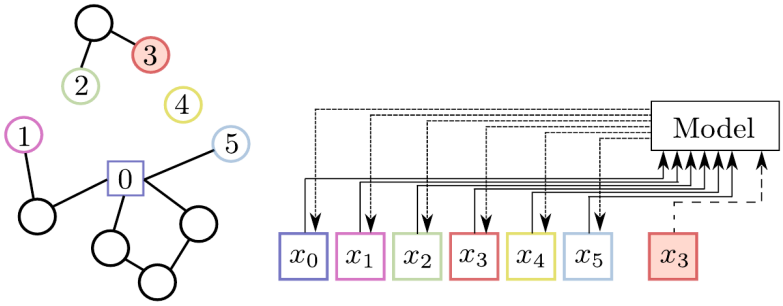
(C) After neural destroy

FIGURE 4.3: Destroy operation based on the edges likelihood heatmap.

In order to guarantee diversification during the process, the edges to remove are not greedily selected taking the ones with lower likelihoods, otherwise every partial solution of the neighborhood would contain the same configuration to repair. Conversely, they are sampled considering the probability density function of the normalized values of the heatmap: the more likely an edge belongs to the optimal solution according to the model, the less likely it is removed.

4.1.2 Neural Destroy & Repair

The neural model employed for the repair operation is adapted from the work of Hottung & Tierney [27]. Given the incomplete solution from the destroy phase, the model stochastically selects one of the unconnected nodes and outputs a probability distribution over all partial tours this customer can be linked to, meaning that an action consists in joining the ends of two incomplete tours. This process is repeated until a feasible solution (i.e., only containing complete routes) is reached. A visual representation of an iteration of the process is provided in figure 4.4.



Source: Hottung & Tierney [27]

FIGURE 4.4: An example incomplete solution and the associated model input in the neural repair model.

The policy learned by the model adapts to the specific method used for deteriorating the current solution. The peculiar aspect of this work consists in performing a *neural hybridization* between the DL-based approach proposed in section 4.1.1 and the just described neural repair heuristic. This translates in parameterizing the architecture in figure 3.2 on the decisions taken by the GNN during the destroy operation.

4.1.3 Adaptive Neural Large Neighborhood Search

Up to now, we have only taken into consideration the case in which the exploration of the neighborhood of the current solution is led by a single pair of destroy and repair operations, however, nothing prevents us from using multiple ones within the same environment.

Each destroy/repair procedure involved has an associated weight, which determines how often the method is called during the search, and is adjusted at run-time according

to the observed effects, to favor the ones most suitable for the instance into consideration. This procedure selection is performed using a "roulette wheel" principle: given w_x the weight associated to a destroy/repair couple, the corresponding probability of being selected during the search process is $p_x = \frac{w_x}{\sum_{i=1}^{n} w_i}$. The weight adjustment, instead, is calculated according to the following formula, where $\alpha = 0.2$ is the exponential moving average factor, $c(x)$ and $c(x^t)$ are respectively the cost of the current solution and the cost of the candidate one, and τ is the time needed to perform the iteration:

$$w_x = w_x \cdot (1 - \alpha) + \frac{c(x) - c(x^t)}{\tau} \cdot \alpha$$

This extension of the original LNS framework takes the name of *Adaptive Large Neighborhood Search* (ALNS), and has the advantage of being more robust. Indeed, while in pure LNS we have to select a destroy/repair procedure that is expected to work well for a wide range of instances, in ALNS we can afford to include methods that only are suitable in some cases, since the adaptive weight adjustment will ensure that these heuristics will seldom be used on instances where they would be ineffective.

In particular, we are interested in an *Adaptive Neural Large Neighborhood Search* (ANLNS), determining what are the effects of mixing traditional destroy methods, like the ones suggested in section 2.3.2, with our GNN approach from section 4.1.1, matching each destroy operation with the neural repair counterpart parameterized on its decisions.

4.2 NeuRouting: A Hybridization Framework

While previously we have explained the details of the approaches we have adopted, in this section we briefly present the main components characterizing the architecture of *NeuRouting*, our framework built from scratch for comfortably making experiments aimed at answering the open research questions at the intersection between DL, LNS, and VRP.

4.2.1 Instances & Solutions

The first and essential component of any CO tool is the class that represents the instance of the problem of interest, in our case CVRP. Even though our research considers only this formulation, it is easy to extend the current implementation by inheriting its current properties, and placing additional constraints (e.g., time windows) to address other vehicle routing variants (e.g., VRP-TW).

Beside specifying all the properties regarding an instance, there exist easier ways to create it. The main modalities are two: generating the instance according to a specific probability distribution, like explained in section 5.2.1, or loading a particular instance from a textual file following the *TSPLib* format[1]. The former approach is particularly useful when there is need to create a lot of instances with common characteristics (e.g., during training), while the latter when we want to consider problems taken from the literature.

One of the critical design choices used in our framework concerns the clear decoupling between an instance and its feasible solutions. This is due to the fact that while a solution is always related to a single instance, for the same instance there exist a huge amount of solutions, each characterized by its own routes and the actual cost, which determines its quality.

Furthermore, since we need to rely on a MILP solver in order to perform some specific operations (i.e., exact reparation of a partial solution), we also provide a conversion tool for expressing the CVRP instance as an objective function subject to a set of constraints compatible with the SCIP free optimizer. This feature can be considered the bridge between *NeuRouting* and the configurable approaches presented in section 3.2, since it makes the former already prone to be extended with the latter.

4.2.2 Solvers & Environments

The component in charge of outputting a solution when given in input an instance of the problem is called *solver*. It does not include any particular property because it represents a general level of abstraction in which even the traditional approaches we use in section 5.4 can be identified with. From an implementation point-of-view, it is an *interface*

[1] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/DOC.PS`

whose methods to override are `reset` and `solve`. The former manipulates the input instance to make it compatible with the actual solver adopted, the latter defines how the solution search should be performed, optionally specifying a time and/or a maximum number of iterations limit.

Inheriting the general structure of a solver, we can define more specific *environments*, embedding the algorithmic workflow of a heuristic approach inside the `solve` method. In our case concerning LNS specifically, we delegate the construction of the initial solution to the `reset` method, and we improve it following the iterative procedure summarized in the pseudocode 1.

Algorithm 1: Large Neighborhood Search

Input: Instance χ, time limit τ, neighborhood size n, percentage p.
Output: The incumbent solution σ^*.

```
start = clock()
σ = σ* = reset(χ)                    # find initial solution
while clock() − start ≤ τ do
    ψ = neighborhood(σ, n)           # generate copies
    ψ = destroy(ψ, p)                # destroy phase
    ψ = repair(ψ)                    # repair phase
    if min(cost(ψ)) < cost(σ*) then
        σ* = ψ[arg min(cost(ψ))]     # update incumbent solution
    end
    if criteria(ψ, σ) then
        σ = ψ[arg min(cost(ψ))]      # update current solution
    end
end
```

It is important to notice that the proposed algorithm is completely agnostic with respect to the operations involved, indeed, both `destroy` and `repair` are components independent of the environment they are attached to. This design follows the famous *strategy pattern* [20], which enables to encapsulate algorithms so that they can be swapped each other to carry out a specific behavior.

The extensibility of this *LNS environment* is not limited to the ease it provides in adding new destroy/repair methods, but is also rooted in the high configurability of its specific aspects. For instance, the default `criteria` for updating the current solution is the same used for updating the incumbent one, however, if we want to integrate aspects peculiar to other metaheuristics (e.g., stochastically allowing worsening solutions as contemplated in simulated annealing), we only need to override this single function.

4.2.3 Destroy/Repair Operations

In the previous section we have emphasized the fact that any operation, being it for destroying or repairing a solution, is conceptually detached from the LNS environment where it is employed, so that it is possible to use it also as a stand-alone algorithm. Take for instance the neural destroy heuristic of section 4.1.1, given a feasible solution and a destruction percentage, it produces a partial assignment like the one in figure 4.3c.

However, using destroy without repair (or vice versa) does not provide any real advantage, since the benefits of using one can only be appreciated when the other is contrarily applied. This is especially evident in presence of neural-based methods, indeed we load different weights for the repair policy depending on which destroy methodology and percentage has been used for the training procedure, otherwise it would be ineffective.

In the face of these observations, we can motivate our design choice of injecting destroy and repair components in an ALNS setting as pairs of complementary operations, called *LNS operators*, rather than two distinct sets of independent algorithms.

4.2.4 Evaluator

We have presented all the tools needed to perform our investigation of the best LNS operator or the best combination of them. The only missing component is the one responsible for fairly evaluating the different solvers made available within *NeuRouting*. This is represented by the *evaluator*, from which the method `compare` can be called specifying a set of instances that all the solvers taken into consideration must face.

The main advantage of using the evaluator lies in the possibility of specifying, at the same time for all the involved solvers, the identical temporal and algorithmic limitations. Moreover, it enables to repeat the same experiments for multiple runs, in order to obtain a more reliable estimation of the effectiveness of each approach with respect to the considered set of instances.

Regarding the produced results, for each solver a different table is generated, in which the rows correspond to the instances, and the columns to the statistics averaged over the executed runs.

Chapter 5

Experimental Study

In this chapter we want to evaluate the capabilities of our NLNS both on instances of the same size of training used as a *test* set, and on larger ones to understand their generalization capabilities. In addition to the different composable NLNS environments, we perform the same benchmark also on some traditional VRP solvers in order to obtain a comparison also with external tools.

Section 5.1 explains in detail how the experimental study is conducted, section 5.2 how NLNS environments are trained, section 5.3 investigates the performance of all the composable NLNS environments, section 5.4 takes the most promising candidates from the previous experiments to compare them with traditional solvers for VRP, and in section 5.5 we test the best approaches on larger instances from the literature, with $100 < n < 300$. Finally, section 5.6 provides a general discussion about the results obtained.

5.1 Experimental Setup

Since the *random* method is the one producing the worse results according to the evaluation performance in figure 5.1, we have decided not to investigate it.

In order to fairly evaluate the different configurations, we always use the same set consisting of 50 instances, independently generated considering the $n = 50$ and the $n = 100$ customers case, following the distribution described in section 5.2.1.

For each instance, the corresponding solution can be searched within a time limit, set to 60 seconds, with the additional restriction of a maximum number of iterations, set to 100 steps. This choice is motivated by the inner difficulty to compare run times;

indeed, the programming language used for the implementation (e.g., Python vs C++), but also the hardware where the algorithms are executed (e.g., CPU vs GPU) can make a huge difference in terms of performance, without appropriately describing the real effectiveness of the inspected approach.

All the trainings and the experiments are carried out on tone of the nodes of our university cluster, running Debian 4.19 and equipped with an NVIDIA GeForce RTX 2080 Ti on the GPU side and with a quad-core 2.2Ghz KVM on the CPU side. In order to obtain more reliable results, the statistics are averaged over 5 runs, and, while the instances from the literature are individually analyzed, for the 50 generated ones we compute the mean to provide a more immediate quality measure.

Among the many configurations that will be presented, the rows corresponding to the ones of particular interest for answering the research questions motivating our work are highlighted in light gray, while the best result in each group is shown in bold.

5.2 NLNS Training

Each neural operator we want to employ inside our NLNS setting, being it a destroy or a repair method, requires a peculiar training process before being used, otherwise, its performance would be nothing different from a trivial random exploration of the solutions space.

NeuRouting provides a flexible solution to this need: during the implementation of an algorithm requiring a DL model, we describe how a training step on a single batch should be performed and, when launching the training, we must specify which opposite operation (i.e., a destroy operation if the neural method is a repair and vice versa) should be used.

5.2.1 Instance Generation

Whatever type of training we decide to opt for, the main ingredient to perform it is always data. In the context of VRP, there is no commonly agreed dataset currently available in literature beyond some benchmark collections[1], which, however, comprise

[1] `http://akira.ruc.dk/~keld/research/LKH-3/BENCHMARKS/`

only a maximum of few hundreds of instances. Therefore, the most adopted solution is to generate the positions of the depot and the nodes, along with their demands, according to some distribution so that it can be replicated in other works.

Our work follows this same mechanism on one of the most adopted data generation algorithms, which was first proposed by Nazari *et. al.* [42]. Specifically, it consists in uniformly sampling the x and y coordinates in the $[0, 1]$ interval, and the demands with an integer value between 1 and 9 included. The vehicle capacity depends on the number of customers to serve, in particular, to 10 clients corresponds 20, to 20 corresponds 30, to 50 corresponds 40, and to 100 corresponds 50.

We have decided to focus our attention on the instances with at least 50 customers, which, for their complexity, are the ones where the exploration of a large neighborhood can be more beneficial with respect to the use of traditional methods.

As it is always good practice to proceed during a machine learning training, we create two separate datasets, one for *training* the model, and one for *evaluating* its performance across the epochs on never seen data. These datasets, containing respectively 100000 and 100 instances, are maintained across all the executed trainings, in order to evaluate the different combinations of destroy and repair operations exactly in the same conditions.

5.2.2 Training Setting

In order to be coherent with the works we base our neural methods on [27, 34], we have decided to keep both the hyperparameters of the models and their training implementation as they are described in the original papers.

Neural Destroy

The *Residual Gated GCN* comprises 30 layers, has a hidden dimension for each node and edge of 300, uses *mean* as the aggregation function, and the final classifier is a Multi-Layer Perceptron with 3 layers (look at figure 2.6 for a visual reference). The training is performed in a supervised manner as described in the original work of Joshi *et. al.* [30], however, the traditional solver to imitate is not *Concorde*, peculiar to TSP, but *LKH*, which addresses also more general routing problems like CVRP.

Neural Repair

Differently from the neural destroy, which do not make use of the opposite operator while learning, the neural repair model printed in figure 3.2 is trained using reinforcement learning adapting its decisions to the specific method selected for destroying the solution. In particular, the reward is measured as the difference between the total tour length of the repaired solution and the partial tour length of the destroyed solution.

In order to stabilize the learning process, a *critic* model is trained, in alternation with the *actor* model (i.e., the one who executes the policy), to minimize the mean squared error between its prediction for the cost of repairing the partial solution and the actual cost when using the most recently learned policy.

5.2.3 Performance

We focus our attention on the performance of the different destroy and repair combinations during training to determine which ones are the most promising. Following the approach of [27], we have tried both 15% and 25% as degrees of destruction when the number of customers is 50, and 10% and 20% when it is 100. The number of epochs, after an experimental observation of the progresses, has been set to 50, while the batch size to 256 as reported in the reference papers.

The training time of each NLNS configuration is linked to the hyperparameters and to the individual destroy/repair components it comprises. The evaluation step is executed at the end of each epoch in an environment analogous to the one the destroy and repair methods are supposed to operate in. Specifically, we run a LNS for an amount of iterations equals to the number of customers, using as neighborhood dimension (i.e., the number of solutions destroyed and repaired in parallel at each iteration) the batch size.

Table 5.1 highlights the most crucial aspects for estimating the training time. In particular, *neural* destroy is computationally heavier than the other non-neural alternatives, requiring between 4 to 10 times more and presenting a higher sensitivity to the instance size n. Conversely, a significant difference among the other methods can be noticed only when the number of nodes $n = 100$ and the destroy percentage $p = 0.2$. Times required

Destroy	Repair	Epochs	Batch	p	n	Time
random	neural	50	256	15%	50	3h 23m
point	neural	50	256	15%	50	2h 55m
tour	neural	50	256	15%	50	2h 30m
neural	neural	50	256	15%	50	12h 22m
random	neural	50	256	25%	50	3h 55m
point	neural	50	256	25%	50	3h 43m
tour	neural	50	256	25%	50	3h 25m
neural	neural	50	256	25%	50	12h 59m
random	neural	50	256	10%	100	4h 28m
point	neural	50	256	10%	100	3h 42m
tour	neural	50	256	10%	100	3h 5m
neural	neural	50	256	10%	100	1d 18h 7m
random	neural	50	256	20%	100	6h 15m
point	neural	50	256	20%	100	5h 48m
tour	neural	50	256	20%	100	4h 52m
neural	neural	50	256	20%	100	1d 19h 58m

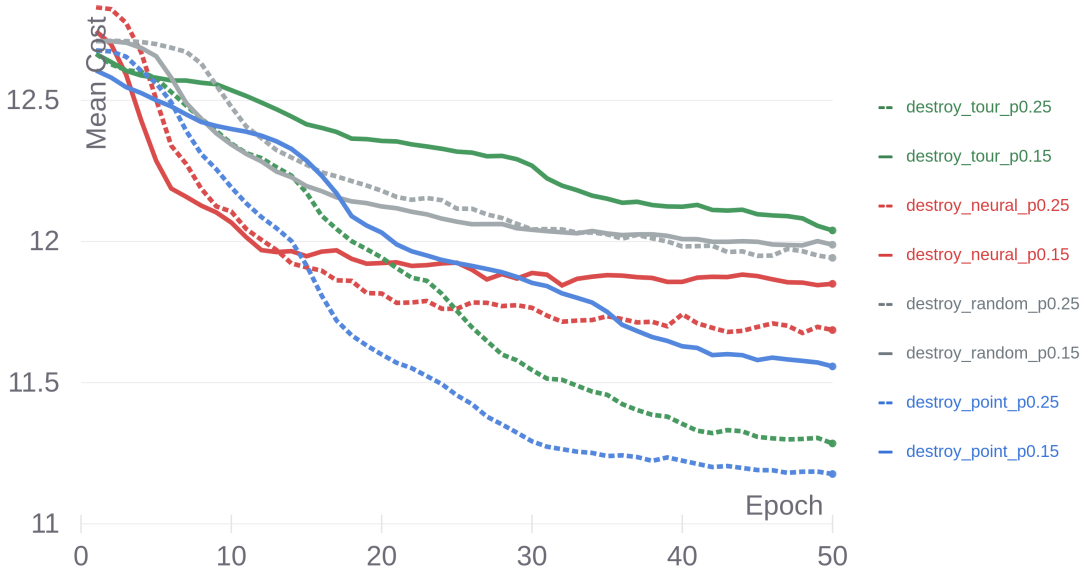
TABLE 5.1: Training time of the different NLNS combinations when varying the instance size and the destroy percentage.

for training must be read considering the need to only perform them once for each combination, because then we can directly load the learned weights inside the corresponding models and make inferences within few seconds.

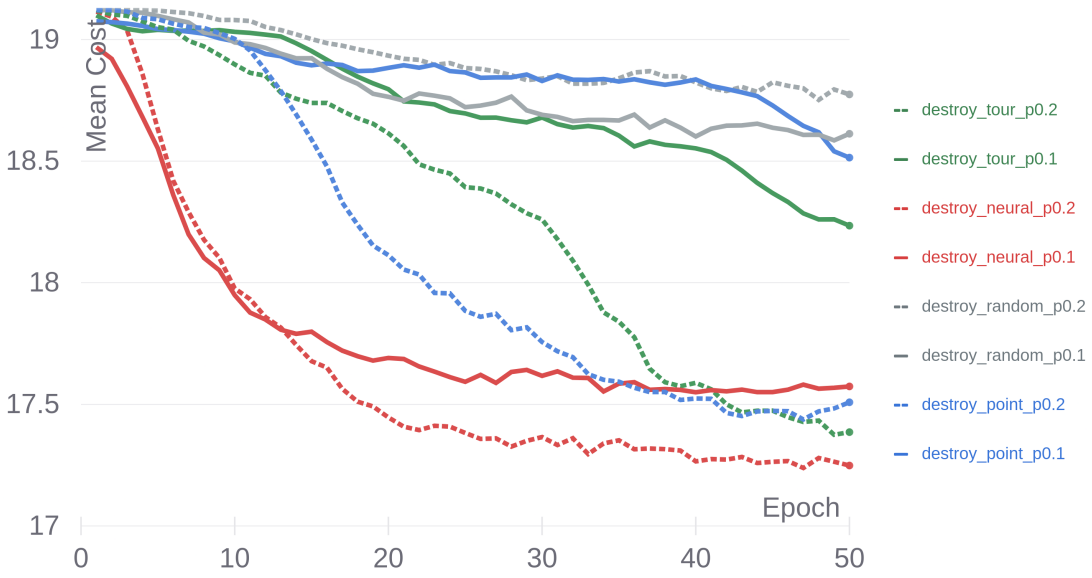
The plots in figure 5.1a and 5.1b describe, respectively for $n = 50$ and $n = 100$, the evolution of the mean cost of the validation instances as the repair policy is updated during the training epochs.

As we could expect, when coupled with the *random* destroy, the neural repair operation is limited in its learning capabilities probably due to the lack of recognizable patterns in the partial solution. Conversely, when the destroy operation exhibits some locality features, like in the *point* or the *tour* destroy, the model can more easily improve its policy, especially when the percentage of solution destroyed is higher.

Mention apart goes to the less human interpretable conjunction between the neural



(A) *NLNS* validation instances mean cost, $n = 50$



(B) *NLNS* validation instances mean cost, $n = 100$

FIGURE 5.1: Comparison of the efficiency of different destroy methods with their respective neural repair policy across the training epochs, using as reference the mean cost of the validation instances. Same color corresponds to the same destroy method, same line style to the same destruction percentage.

destroy and the neural repair procedures, which, while presenting slightly worsen results than the others when $n = 50$, outperforms the competition when $n = 100$. In particular, one of the most impressive outcomes is the huge gap between the *neural* destroy and its alternatives considering the 10% destruction case.

5.3 NLNS Evaluation

Our first study consists in evaluating the potentialities of the different NLNS environments we can build combining the previously presented methods, namely: *point* and *tour*, described in section 2.3.2, plus our *neural* option of section 4.1.1 as regards the destroy; *exact*, relying on the optimal solution of the sub-problem, *greedy*, adopting a heuristic approach, and *neural*, based on the work of [27], for the repair.

5.3.1 Neural Destroy

Since we have defined a neurally-guided destroy operation, we want to analyze the benefits of using it rather than using one of the other available methods when the repair operation is led by an exact solver, as proposed in [1, 51]. Our choice for the optimization suite falls on *SCIP 7.0.3*.

Since the exploration phase is performed during the repair operation, where the current partial assignment is solved until optimality, the *size* of the neighborhood of partial solutions, generated from the incumbent one using the destroy method, consists of only 4 candidates. We have not limited it to only one in order to introduce a little diversification also in these environments, like it is present in the ones treated by the sections 5.3.2 and 5.3.3.

The degree of destruction p is selected so that the resulting sub-problem is solvable by SCIP in a reasonable amount of time (i.e., less than 10 seconds out of the 60 available).

Destroy	**Repair**	**Size**	**p**	**n**	**Cost**
point	exact	4	15%	50	**10.7953**
tour	exact	4	15%	50	10.8934
neural	exact	4	15%	50	10.8387
point	exact	4	10%	100	16.5232
tour	exact	4	10%	100	16.5326
neural	exact	4	10%	100	**16.4245**

TABLE 5.2: Comparison of the different destroy methods when using an exact reparation. The statistics are computed averaging the results of all the instances taken into consideration.

Results in table 5.2 show that the number of customers in the instance makes a huge difference in the effectivity of the *neural* destroy operation. Indeed, while in the $n = 50$ case just exploring regions of space leads to better solutions, in a larger setting of $n = 100$ customers, exploiting the weaknesses in the current routes has a greater impact.

5.3.2 Neural Destroy & Repair

In the last section, repairing using SCIP represents the actual bottleneck of execution for the algorithm, because either the running time is too long for a single iteration, or it is not likely to improve the incumbent solution if p is too small. For this reason, we analyze the impact of a *neural* repair and of a *greedy* algorithm in place of it.

In this case, the exploration of the neighborhood is performed thanks to the destroy method, in fact, given the same partial assignment, both the ANN and the classic heuristic will (almost) always return the same repaired solution. This means we need to provide the repair procedure different partial solutions, which consequently will be reconstructed in different ways leading to distinct solutions. That is why all the destroy operations include a stochastic component, as we detailed in sections 2.3.2 and 4.1.1.

The degree of destruction p, as usually done in ML for hyperparameters, is chosen according to the validation performance in figure 5.1.

With regard to the neighborhood size, we have opted for 256, which is the nearest power of two to the original choice amounting to 300 reported in the reference paper [27]. This small change aims to favor the parallelism of the operations involving the GPU, namely the *neural* ones, without affecting the others.

Statistics in table 5.3 show that neural reparation is always better than a traditional greedy algorithm, irrespective of the number of nodes in the instance taken into consideration but also of the destroy methodology. Considering these outcomes, from now on we will only take this method of reconstruction into consideration.

While the situation where this gap is more pronounced is when the destroy method is guided neurally, this is not the best companion for our neural repair operation. Indeed, both in the $n = 50$ and in the $n = 100$ scenarios, *point* turns out to produce the best solutions.

Destroy	Repair	Size	p	n	Cost
point	greedy	256	25%	50	10.8570
point	neural	256	25%	50	**10.7602**
tour	greedy	256	25%	50	10.9276
tour	neural	256	25%	50	10.8501
neural	greedy	256	25%	50	10.8866
neural	neural	256	25%	50	10.8227
point	greedy	256	20%	100	16.0970
point	neural	256	20%	100	**15.9700**
tour	greedy	256	20%	100	16.2606
tour	neural	256	20%	100	16.1122
neural	greedy	256	20%	100	16.2375
neural	neural	256	20%	100	16.0360

TABLE 5.3: Comparison of repairing using a greedy approach *vs* a neural one, varying the destroy method. The statistics are computed averaging the results of all the instances taken into consideration.

5.3.3 Adaptive Neural Large Neighborhood Search

Whenever different approaches aiming to accomplish the same task prove to be promising, the next obvious step is to try to merge them inside the same environment. In our case, this means using an ALNS setting, where multiple destroy operators and their respective neural repair counterparts collaborate in the neighborhood exploration, as described in section 4.1.3.

Analyzing table 5.4, we can notice, especially in the $n = 100$ case, a general improvement of the solutions quality when pairing operators with respect to when using them alone.

The contribution of the neural destroy operation, as reported in section 5.3.1, has more benefits when the complexity of the instance (i.e., its number of customers) is higher. Indeed, while paring *point* with *tour* gives worse results than only using *point*, when *neural* is involved, either only with the former or in addition to both, solution quality always improve.

Destroy	Repair	Size	p	n	Cost
point+tour	neural	256	25%	50	**10.7315**
point+neural	neural	256	25%	50	10.8471
tour+neural	neural	256	25%	50	10.8188
point+tour+neural	neural	256	25%	50	10.7801
point+tour	neural	256	20%	100	16.0315
point+neural	neural	256	20%	100	**15.9338**
tour+neural	neural	256	20%	100	16.0523
point+tour+neural	neural	256	20%	100	15.9369

TABLE 5.4: Comparison of the different combinations of destroy methods in an ALNS setting performing the repair operation neurally. The statistics are computed averaging the results of all the instances taken into consideration.

Even if the differences among the different approaches is not huge in terms of absolute value, we should remember that the cost is computed on a unitary map, and, therefore, in a real-world scenario it should be scaled on the size of the geographic area taken into account.

5.4 Comparison to Traditional Solvers

Up to now, we have only compared the potentialities of our NLNS environments making them compete with each other. In order to give a broader perspective, this section is intended to challenge the solvers which are traditionally employed to solve VRP: the SCIP optimizer identified with the name *milp*, Google OR-Tools dedicated framework for VRP recognizable in *or-tools*, and LKH heuristic approach (*lkh*).

These traditional solutions can only be evaluated considering the time aspect, since the solving procedures are too heterogeneous to be configured with coherent parameters, therefore, we keep the 60 seconds time limit also for them. The gap in percentage is computed with respect to the considered NLNS configuration both for $n = 50$ and $n = 100$.

According to table 5.5, LKH confirms to be the best solver available for VRP irrespective of the number of nodes in the instance. Conversely, the performances of

Solver	n	Cost	Gap
point+tour → neural	50	10.7315	0.00%
milp	50	11.9540	11.39%
or-tools	50	10.7373	0.05%
lkh	50	**10.6170**	−1.07%
point+neural → neural	100	15.9338	0.00%
milp	100	25.8404	62.17%
or-tools	100	16.3595	2.67%
lkh	100	**15.7762**	−0.99%

TABLE 5.5: Best NLNS approach *vs* traditional VRP solvers. The statistics are computed averaging the results of all the instances taken into consideration.

OR-Tools dedicated routing solver, in line with our most promising approaches when $n = 50$ customers, tend to deteriorate when $n = 100$, with a percentage gap between the 2% and the 3%. SCIP optimizer, as we could expect, cannot compete with the rivals, especially when asked to solve instances containing 100 customers.

These observations seem to suggest that NLNS can give the highest benefits with respect to traditional approaches when dealing with difficult instances. However, we have not yet studied scenarios with which it is not comfortable with, namely when the number of nodes in the problem to solve is different from the number nodes it has been trained with.

5.5 Larger Instances

This last section of the experiments is intended to analyze the performance of the best destroy/repair combinations for $n = 50$ and $n = 100$, on instances from the literature tougher than the ones with which our operators are trained with. Our aim is to check whether they are still effective or not, because, in the former case, this would mean we can avoid the heavy training phase we have described in section 5.2.

Specifically, we have selected ten instances with a variable number of customers $100 < n < 300$ from the work of Uchoa *et al.* [55]. The naming convention used to

describe an instance consists in specifying the number of nodes (depot included) after the letter n, and the optimal number of vehicles after the letter k.

Figure 5.5 compare the solution quality produced by NLNS with respect to LKH, OR-Tools, and the known optimal cost. In line with the previous results, our approaches are generally not as efficient as the state-of-the-art LKH solver, despite reaching results pretty similar to it when $n < 200$, and they beat OR-Tools on 8 out of 10 instances.

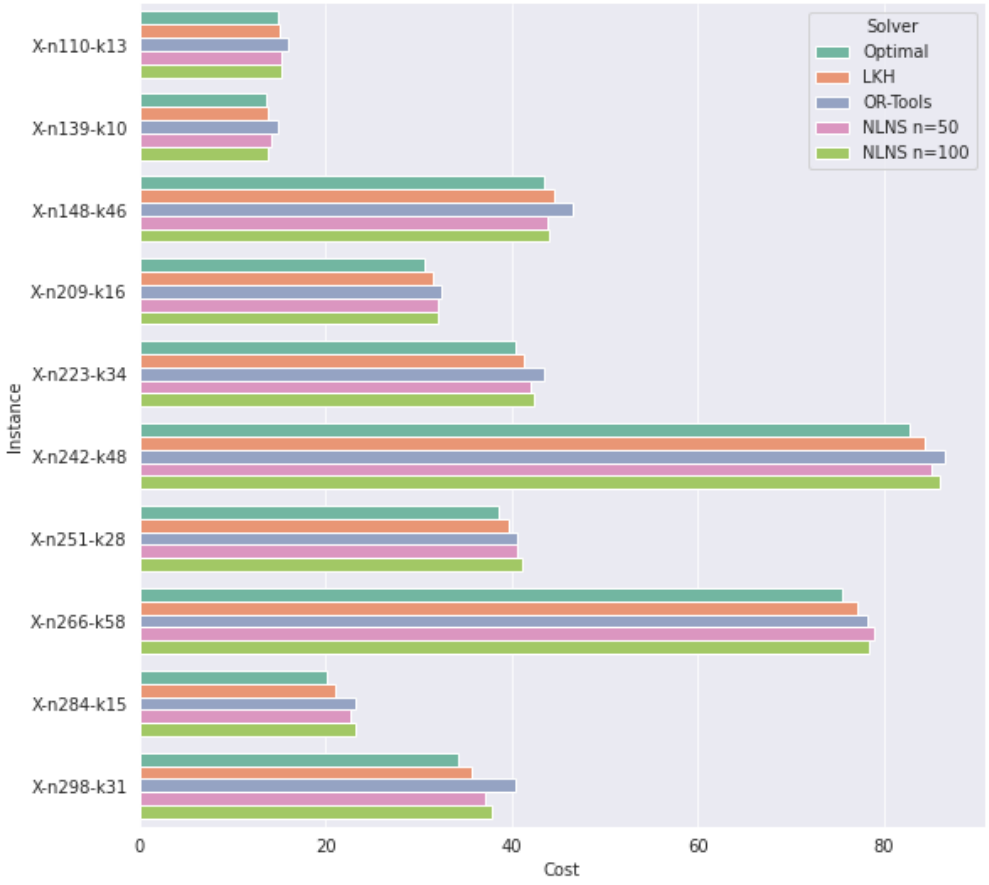


FIGURE 5.2: Performance of the best traditional and NLNS based solvers on instances, from the literature, with a number of nodes $100 < n < 300$.

Notably, the NLNS $n = 100$ approach tends to perform better than NLNS $n = 50$ for $100 < n < 220$, while the opposite occurs when the number of nodes goes toward 300. This seems to suggest that the neural destroy method, which is employed only in the former, is sensitive to the number of nodes and is less effective the more it departs from the training one. This issue can be further investigated following a *transfer learning* technique as suggested by Joshi *et al.* [29] for TSP.

5.6 Discussion of Results

A broader view on the produced results can be obtained analyzing jointly tables 5.1, 5.2, 5.3, 5.4, and 5.5. Despite the small amount of instances (i.e., 50) and the limited resolution time (i.e., 60 seconds) considered in order to try lots of different configurations, we are able to provide some general observations.

The first consideration is about "when" it is meaningful to use NLNS, indeed, while there are no major benefits with respect to traditional solvers when $n = 50$, the higher complexity of the $n \geq 100$ cases makes it interesting to employ them.

Another important aspect deals with the "where" leveraging ML algorithms is beneficial. Their effectiveness is undisputed as regards the repair phase, nevertheless, in the destroy one, improvements related to its usage only arises in an ALNS setting. This is probably due to the fact that this is the environment where there is the better trade-off between large neighborhood exploration and the exploitation of the weaknesses of the current solution.

Eventually, there is no certain answer on "if" it is worth to use NLNS approaches for VRP. On the one hand, LKH remains the solver to beat in most of the cases, and the training time needed to obtain a NLNS environment is high, especially in the completely neural configuration. However, on the other hand, results are promising, because the negative gap with respect to the specialized LKH solver ($\approx 1\%$) is small if compared to the positive one achieved on OR-Tools and SCIP (respectively $\approx 3\%$ and $\approx 62\%$); moreover, the generalization capabilities are good even when the number of nodes triplicates, suggesting the heavy training procedure does not need to be retaken every slight change of the number of nodes in an instance.

In any case, we believe the improvement capabilities of neural approaches are far beyond the ones of traditional solvers, since the research is still in its early stages for the former, while it has been consolidated through more than 60 years for the latter. This work itself provides a lot of research directions that can be pursued starting from the discussed attempts and thanks to which the gap with the current state-of-the-art can be narrowed if not overcome.

Chapter 6

Conclusions

We have finally reached the end of this work. It should probably be considered more like a beginning, because in an attempt to answer the questions we started from, even more risen to our attention. This is probably normal, because the ambitious goal of applying ML to CO, born few years ago, is in its early stages yet. For this reason, while in section 6.1 we summarize all the relevant aspects of the study, in section 6.2 we suggest some research directions to pursue based on it.

6.1 Summary of Contributions

Within the wide and complex world of CO, we have focused our attention on the CVRP, analyzing the various ML techniques available in literature to face it. Among the others, we have been interested in exploring more in-depth how to perform an integration between ML and the LNS metaheuristic.

In particular, we have studied the outcomes of applying neural-based operations either partially or totally during the main phases of LNS, showing benefits and limitations of each kind of usage. While for the repair method we have relied on the same attention mechanism employed in [27], due to the lack of a destroying alternative peculiar for CVRP, we have proposed our own taking inspiration from the GNN architecture presented in [34].

Our results show that considering DL-enhanced approaches is more effective with respect to using traditional solvers only when the number of nodes in the instance is at least 100. However, at the same time, it tends to deteriorate, especially in the GNN case, as this number moves away from the size used for training. In any case, this degradation

of performance is not rapid, in fact we have been able to obtain good results up to 300 nodes.

VRP, besides their inherent difficulty, present a wide variety of facets, which make them probably unsuitable to be solved using a single universal approach. To make a comparison with a simpler problem, we can mention the most well known of computer science: sorting. There exist a huge number of algorithms, each with its own efficiency to solve it, however, standard libraries of the established programming languages (e.g., C++, Python) do not choose one in particular, rather, they implement a hybrid between some of them, which switch to the most suitable depending on the amount of data and on the progress of the algorithm. In our case, this hybridization is the ALNS environment, while the single sorting routines it exploits are the heuristics (neural or not) used as destroy or repair operations.

The state-of-the-art VRP solver, LKH, in almost all the experiments, turned out to beat our NLNS, despite the low percentage gap with respect to it. Nevertheless, there are also cases where the opposite occurred, in particular the first three instances in section 5.5. The relevance of this specific observation has a broader meaning: even if the proposed neural operators have demonstrated effective but generally not as good as the most powerful traditional heuristics, this does not mean they are unbeatable, and, for this reason, it is useful to adapt and explore other methods, relying on different architectures, in place of the ones we have tested, in order to try overcoming them. Not by chance, *NeuRouting* is designed to favor exactly this kind of contributions, providing an easy-to-use and heuristic-agnostic framework.

6.2 Future Work

The end of the previous section might have already suggested the most straightforward research direction to pursue: trying other DL-based operators in place of the current available destroy or repair alternatives.

Another interesting aspect to analyze comes from the proverb “well begun is half done”. In section 2.3.1 we have detailed how the initial solution – i.e., the one from which the LNS starts –, is constructed. However, it is usually of poor quality and the path to become nearly optimal can be way harder than if we have begun from a better

one. The algorithms through which obtaining a good initial solution can be designed following different procedures. For instance, we can use one of the constructive approaches in section 3.1. In any case, it is always important to pay attention at maintaining a good trade-off between the time spent to determine this *warm start* and the one dedicated to improve it, to which should probably be given the highest priority.

The metaheuristic we have focused upon is LNS, because we believe it is one of the most promising techniques to face VRP. Notice that its adoption does not prevent it to be enhanced with other metaheuristics, like *Simulated Annealing* and *Tabu Search*, therefore this kind of integrations can be further explored.

Last but not least, we should remind that within the big VRP family presented in section 2.2.1, our experiments address in particular the notorious case of CVRP. This means any other variant is suitable to be investigated using the same approaches but different models, and, according to the same principle, also completely different problems for which LNS already demonstrated effective, *scheduling* to mention one, can take inspiration from this work.

Bibliography

1. Addanki, R., Nair, V. & Alizadeh, M. *Neural Large Neighborhood Search* in *Learning Meets Combinatorial Algorithms at NeurIPS2020* (2020).
2. Alinson S. Xavier & Qiu, F. *ANL-CEEESA/MIPLearn v0.1* 2020.
3. Bahdanau, D., Cho, K. & Bengio, Y. *Neural Machine Translation by Jointly Learning to Align and Translate* 2016. arXiv: 1409.0473 [cs.CL].
4. Bello, I., Pham, H., Le, Q. V., Norouzi, M. & Bengio, S. *Neural Combinatorial Optimization with Reinforcement Learning* 2017. arXiv: 1611.09940 [cs.AI].
5. Bengio, Y., Lodi, A. & Prouvost, A. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research* **290,** 405–421 (Apr. 2021).
6. Bianchi, L., Dorigo, M., Gambardella, L. M. & Gutjahr, W. J. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing* **8,** 239–287 (Sept. 2008).
7. Bresson, X. & Laurent, T. *Residual Gated Graph ConvNets* 2018. arXiv: 1711.07553 [cs.LG].
8. Cappart, Q., Moisan, T., Rousseau, L.-M., Prémont-Schwarz, I. & Cire, A. *Combining Reinforcement Learning and Constraint Programming for Combinatorial Optimization* 2020. arXiv: 2006.01610 [cs.AI].
9. Cappart, Q. *et al. Combinatorial optimization and reasoning with graph neural networks* 2021. arXiv: 2102.09544 [cs.LG].
10. Carion, N. *et al. End-to-End Object Detection with Transformers* in *Computer Vision – ECCV 2020* (eds Vedaldi, A., Bischof, H., Brox, T. & Frahm, J.-M.) (Springer International Publishing, Cham, 2020), 213–229.

11. Chalumeau, F., Coulon, I., Cappart, Q. & Rousseau, L.-M. *SeaPearl: A Constraint Programming Solver Guided by Reinforcement Learning* in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (ed Stuckey, P. J.) (Springer International Publishing, Cham, 2021), 392–409.

12. Chen, X. & Tian, Y. *Learning to Perform Local Rewriting for Combinatorial Optimization* in *Advances in Neural Information Processing Systems* (2019).

13. Cook, S. A. *The complexity of theorem-proving procedures* in *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71* (ACM Press, 1971).

14. Croes, G. A. A Method for Solving Traveling-Salesman Problems. *Operations Research* **6,** 791–812 (Dec. 1958).

15. Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B. & Song, L. *Learning Combinatorial Optimization Algorithms over Graphs* in *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Curran Associates Inc., 2017), 6351–6361.

16. Dantzig, G. B. & Ramser, J. H. The Truck Dispatching Problem. *Management Science* **6,** 80–91 (Oct. 1959).

17. Dantzig, G. B. *Origins of the simplex method* June 1990.

18. Desrochers, M. & Laporte, G. Improvements and extensions to the Miller-Tucker-Zemlin subtour elimination constraints. *Operations Research Letters* **10,** 27–36 (Feb. 1991).

19. Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y. & Rousseau, L.-M. in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research* 170–181 (Springer International Publishing, 2018).

20. Gamma, E., Helm, R., Johnson, R. E. & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

21. Gasse, M., Chételat, D., Ferroni, N., Charlin, L. & Lodi, A. *Exact Combinatorial Optimization with Graph Convolutional Neural Networks* in *Advances in Neural Information Processing Systems 32* (2019).

22. Gonzalez-Feliu, J. *Models and Methods for the City Logistics: The Two-Echelon Capacitated Vehicle Routing Problem* PhD thesis (May 2008).

23. Goodfellow, I., Bengio, Y. & Courville, A. *Deep Learning* (MIT Press, 2016).

24. Helsgaun, K. An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems. en (2017).

25. Helsgaun, K. General k-opt submoves for the Lin–Kernighan TSP heuristic. *Mathematical Programming Computation* **1,** 119–163 (July 2009).

26. Hornik, K., Stinchcombe, M. & White, H. Multilayer feedforward networks are universal approximators. *Neural Networks* **2,** 359–366 (Jan. 1989).

27. Hottung, A. & Tierney, K. *Neural Large Neighborhood Search for the Capacitated Vehicle Routing Problem* in *24th European Conference on Artificial Intelligence (ECAI 2020)* (2020).

28. Joshi, C. Transformers are Graph Neural Networks. *The Gradient* (2020).

29. Joshi, C. K., Cappart, Q., Rousseau, L.-M. & Laurent, T. *Learning TSP Requires Rethinking Generalization* 2021. arXiv: 2006.07054 [cs.LG].

30. Joshi, C. K., Laurent, T. & Bresson, X. *An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem* 2019. arXiv: 1906.01227 [cs.LG].

31. Jumper, J. *et al.* Highly accurate protein structure prediction with AlphaFold. *Nature* **596,** 583–589 (July 2021).

32. Karp, R. M. in *Complexity of Computer Computations* 85–103 (Springer US, 1972).

33. Kipf, T. N. & Welling, M. *Semi-Supervised Classification with Graph Convolutional Networks* 2017. arXiv: 1609.02907 [cs.LG].

34. Kool, W., van Hoof, H., Gromicho, J. & Welling, M. *Deep Policy Dynamic Programming for Vehicle Routing Problems* 2021. arXiv: 2102.11756 [cs.LG].

35. Kool, W., van Hoof, H. & Welling, M. *Attention, Learn to Solve Routing Problems!* 2019. arXiv: 1803.08475 [stat.ML].

36. Laporte, G., Ropke, S. & Vidal, T. in *Vehicle Routing* 87–116 (Society for Industrial and Applied Mathematics, Nov. 2014).

37. Lombardi, M. & Milano, M. *Boosting Combinatorial Problem Modeling with Machine Learning* in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence* (International Joint Conferences on Artificial Intelligence Organization, July 2018).

38. Lu, H., Zhang, X. & Yang, S. *A Learning-based Iterative Method for Solving Vehicle Routing Problems* in *International Conference on Learning Representations* (2020).

39. Mazyavkina, N., Sviridov, S., Ivanov, S. & Burnaev, E. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research* **134,** 105400 (Oct. 2021).

40. Mirhoseini, A. *et al.* A graph placement methodology for fast chip design. *Nature* **594,** 207–212 (June 2021).

41. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518,** 529–533 (Feb. 2015).

42. Nazari, M., Oroojlooy, A., Takáč, M. & Snyder, L. V. *Reinforcement Learning for Solving the Vehicle Routing Problem* in *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Curran Associates Inc., 2018), 9861–9871.

43. Nowak, A., Villar, S., Bandeira, A. S. & Bruna, J. *Revised Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks* 2018. arXiv: 1706.07450 [stat.ML].

44. *Applications of Combinatorial Optimization* (ed Paschos, V. T.) (John Wiley & Sons, Inc., July 2014).

45. Pisinger, D. & Ropke, S. in *Handbook of Metaheuristics* 99–127 (Springer International Publishing, Sept. 2018).

46. Prouvost, A. *et al. Ecole: A Gym-like Library for Machine Learning in Combinatorial Optimization Solvers* in *Learning Meets Combinatorial Algorithms at NeurIPS2020* (2020).

47. Ramesh, A. *et al. Zero-Shot Text-to-Image Generation* 2021. arXiv: 2102.12092 [cs.CV].

48. Ropke, S. & Pisinger, D. An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. *Transportation Science* **40,** 455–472 (Nov. 2006).

49. Sanchez-Lengeling, B., Reif, E., Pearce, A. & Wiltschko, A. B. A Gentle Introduction to Graph Neural Networks. *Distill.* https://distill.pub/2021/gnn-intro (2021).

50. Shaw, P. in *Principles and Practice of Constraint Programming — CP98* 417–431 (Springer Berlin Heidelberg, 1998).

51. Sonnerat, N., Wang, P., Ktena, I., Bartunov, S. & Nair, V. *Learning a Large Neighborhood Search Algorithm for Mixed Integer Programs* 2021. arXiv: 2107.10201 [math.OC].

52. Stuckey, P. J. in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems* 5–9 (Springer Berlin Heidelberg, 2010).

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

54. *Vehicle Routing* (eds Toth, P. & Vigo, D.) (Society for Industrial and Applied Mathematics, Nov. 2014).

55. Uchoa, E. *et al.* New benchmark instances for the Capacitated Vehicle Routing Problem. *European Journal of Operational Research* **257,** 845–858 (Mar. 2017).

56. Vaswani, A. *et al. Attention is All You Need* in *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Curran Associates Inc., 2017), 6000–6010.

57. Veličković, P. *et al.* Graph Attention Networks. *International Conference on Learning Representations* (2018).

58. Vesselinova, N., Steinert, R., Perez-Ramirez, D. F. & Boman, M. Learning Combinatorial Optimization on Graphs: A Survey With Applications to Networking. *IEEE Access* **8,** 120388–120416 (2020).

59. Vinyals, O., Fortunato, M. & Jaitly, N. *Pointer Networks* 2017. arXiv: 1506.03134 [stat.ML].

60. Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* **8,** 229–256 (May 1992).

61. Wu, Y., Song, W., Cao, Z., Zhang, J. & Lim, A. Learning Improvement Heuristics for Solving Routing Problems. *IEEE Transactions on Neural Networks and Learning Systems,* 1–13 (2021).

62. Wu, Z. *et al.* A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* **32,** 4–24 (Jan. 2021).