University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Computer Science

Size Generalization of Learning-Based Approaches for the Travelling Salesperson Problem

Supervisor
Prof. Fabio Vandin
University of Padova

Master Candidate
Nikola Bulat

Student ID
2073521

Academic Year
2023-2024

I would like to dedicate this thesis to my family and friends.

Abstract

This thesis focuses on learning-based approaches for the Travelling Salesperson Problem (TSP). While these techniques achieve comparable performance to classical solvers on small sizes, they fail to generalize the learned strategies to larger instances. In particular, Graph Neural Network (GNN) architectures with non-autoregressive (NAR) decoding scheme show notably poor performance when applied to graph sizes beyond those seen during training. However, leveraging the efficiency of NAR models through the compositional paradigm has led to some of the top-performing learning-based methods in the literature. Even though such approaches had a lot of success by breaking down the original problem into subproblems, they ignore the potential to combine direct and compositional generalization. We address this by leveraging the direct size generalization improvements in the compositional setting. In particular, our model achieves competitive performance compared to previous methods, having 9 times fewer parameters than the current architecture used in the same framework. Moreover, during our exploration of direct NAR generalization, we also tested SizeShiftReg, a novel regularization method originally designed for the graph classification task. To the best of our knowledge, this is the first application of SizeShiftReg to learning-based methods in the Combinatorial Optimization (CO) domain. We obtain a 7.4% reduction in the average optimality gap on the test dataset, which indicates the potential of SizeShiftReg to improve the direct generalization of learning-based approaches for the TSP. However, further experimentation with significantly more computational resources is required to scale the application of SizeShiftReg to the amount of data used in state-of-the-art models.

Contents

Listing of figures

Listing of tables

Listing of acronyms

AGCN . . . . Anisotropic Graph Convolutional Network

AR . . . . Autoregressive (decoding)

Att-GCRN . . . . . Graph Convolutional Residual Network with Attention Mechanism

CMD . . . . Central Moment Discrepancy

CO . . . . Combinatorial Optimization

CP . . . . Constraint Programming

DP . . . . Dynamic Programming

DPDP . . . . Deep Policy Dynamic Programming

GCN . . . . Graph Convolutional Network

GNN . . . . Graph Neural Network

MCTS . . . . Monte Carlo Tree Search

ML . . . . Machine Learning

MLP . . . . Multi-layer Perceptron

NAR . . . . Non-autoregressive (decoding)

NCO . . . . Neural Combinatorial Optimization

PTAS . . . . Polynomial-Time Approximation Schemes

FPTAS . . . . Fully Polynomial-Time Approximation Schemes

RL . . . . Reinforcement Learning

SL . . . . Supervised Learning

TSP . . . . Travelling Salesperson Problem

UL . . . . Unsupervised Learning

VRP . . . . Vehicle Routing Problem

Notation

This section outlines the notation used throughout the thesis. The listed symbols represent the default notation, and any exceptions will be clearly specified as needed.

Numbers, Arrays, and Functions

x	A scalar
$\boldsymbol{x}$	A vector
$\boldsymbol{X}$	A tensor of ≥ 2 dimensions
$f(x), F(x)$	A function of scalar input
$\mathcal{L}(\theta)$	A loss function, where θ are model parameters

Indexing

x_i	i-th element of vector $\boldsymbol{x}$
$X_{i,j}$	Element (i, j) of matrix $\boldsymbol{X}$

We use the zero-based numbering for the indices. That is, our indices go from 0 to $n - 1$ for an array of size n. For a $n \times m$ matrix, the rows and columns go from 0 to $n - 1$ and 0 to $m - 1$, respectively.

Sets

X	A set
$[a, b]$	The real interval including a and b
$\mathbb{R}$	The set of real numbers
$\mathbb{Z}$	The set of integers

Regular Latin letters (e.g., X, Y) are used to denote general sets. Blackboard bold symbols (e.g., $\mathbb{R}$, $\mathbb{Z}$) are reserved for special sets, such as the set of real numbers or integers.

1 Introduction

In this chapter, we provide a concise introduction to our work. We begin by motivating the problem in general as well as the specific research scope that we are interested in. Following that, we summarize our main contributions. Finally, we provide an overview of the thesis structure.

Motivation

Combinatorial Optimization (CO) focuses on problems where the objective is to optimize a cost (or objective) function by selecting the best subset from a finite set, subject to constraints that determine the feasible solution space [1]. Despite their inherent theoretical complexity, as many CO problems of practical interest are NP-hard, they are often solvable in practice using heuristic methods that provide efficient solutions for specific instances. However, these techniques typically rely on expert knowledge acquired through years of research on specific problems. Furthermore, practical problem instances often involve minor changes in constraints, features, or even a slightly redefined version of the same problem. Recently, Machine Learning (ML) approaches have been proposed to overcome these issues, leveraging their ability to discover common patterns in given problem instances. These approaches could allow faster development of efficient algorithms for practical applications [1, 2].

Among the 21 NP-complete problems identified by [3], ten are decision versions of graph optimization problems, and many others can also be represented using graphs [1]. Moreover, the

2D Euclidean Travelling Salesperson Problem (TSP) has been the most intensely studied NP-hard graph problem in the operations research community. In addition to its significance in applications such as logistics, genetics, and scheduling, the progress made in solving the TSP has also been used as inspiration in solutions to related problems [4]. However, as the number of nodes increases beyond thousands, the running time of even the most advanced solvers, such as Concorde [5], becomes prohibitive for practical applications.

ML offers an alternative approach that leverages data to learn the policy for solving the TSP and its variations. Given the natural graph structure of the problem, the Graph Neural Network (GNN) architecture has been the most successful in this task. Once trained, GNNs provide significant time improvements with respect to their heuristic counterparts [4]. However, GNN approaches have shown competitive performance compared to classical solvers only on graphs with limited number of nodes due to the inherent generalization issue. In fact, learning the underlying structural patterns in the data that scale to large instances has been identified as one of the key challenges in the field [1, 4]. Recently, some approaches have tried solving this issue by training on large instances directly [6, 7]. However, they often require complex models, extensive resources, and longer inference time. More importantly, generalization across sizes presents a fundamental property that ML approaches should obtain.

The authors of [4] describe a unified pipeline that encompasses recent ML approaches for learning the TSP in the literature. They opt for zero-shot generalization from smaller to larger instances due to the time-consuming training on large instances directly. They distinguish between autoregressive (AR) and non-autoregressive (NAR) architectures. NAR approaches produce heatmaps by predicting the probability of each edge belonging to the optimal solution independently. On the other hand, AR approaches model the sequential inductive bias through step-by-step graph traversal. Their analysis focuses on the AR architecture due to its favorable sequential decoding scheme. In contrast, NAR architectures, though faster, scale poorly to larger graph sizes than present in the training data.

The most popular approach to overcome the poor generalization of NAR architectures is the application of the divide-and-conquer paradigm. For example, the authors of [8] pretrain a GNN on small graphs of fixed size (20 or 50). However, instead of directly using the model to produce the heatmap at inference time, they sample a number of subgraphs of the same fixed size, use the model directly on each subgraph separately, and merge the outputs into the heatmap for the original large graph. Finally, they use the Monte Carlo Tree Search (MCTS) to construct the final solution from the merged heatmap. With this simple method, they obtain good generalization performance on graph sizes up to 10000. In fact, their testing setting has been adopted in most of the subsequent works in the literature, which confirms the importance of this paper. However, their pretrained models on graphs of small size have a large number of parameters that, when combined with an expensive MCTS on the heatmaps, could prohibit real-time applications. Moreover, their approach completely neglects the direct generalization of models which leads to a complex model overfitting on a specific graph size.

Our Contributions

We build on top of the work in [4] and [8]. That is, we identify that the approach in [8] uses an overly-complex model (approximately $11{,}000{,}000$ trainable parameters) to train on relatively small graphs of size 50. Moreover, their model focuses specifically on performing well on a fixed graph size without any analysis of its direct generalization performance. We show that having a simpler model with better generalization can be beneficial for three main reasons, i.e., faster inference time, significant memory savings, and the potential to extend the compositional heatmap construction by including multiple levels of composition, sampling graphs outside of training size, and/or including variable graph sampling sizes at inference time. Moreover, we show the possibility of improving the direct generalization of NAR approaches even with the lack of sequential inductive bias. We elaborate on our methods in the following paragraphs.

We begin by analysing the effects of simple architectural changes (model scale, normalization methods, activation functions, depth/width ratio) on the direct generalization performance of NAR architectures. Following that, we apply SizeShiftReg [9] to the CO domain. SizeShiftReg is a regularization method originally designed to improve size-generalization capabilities of GNNs for the graph classification task. It operates by simulating a size shift in the training data and regularizing the GNN to be robust to such a shift. Size shift is obtained using graph coarsening techniques while the regularization aims to minimize the discrepancy between the node embedding distributions of the original graphs and its size-shifted versions. To the best of our knowledge, we are the first to apply SizeShiftReg in the domain of combinatorial optimization. Finally, we adapt the compositional heatmap construction introduced in [8] to work with the Anisotropic Graph Convolutional Network (AGCN) architecture from [4].

We summarize our contributions as follows:

- We extend the analysis of the direct generalization performance of NAR architectures by experimenting with model scale, normalization techniques, activation functions, and depth/width ratio.
- We apply SizeShiftReg [9] to the task of learning the edge probability heatmap for the TSP. Our results show that we can obtain up to 7.4% reduction in the average optimality gap on the test dataset while improving (or at least maintaining) the performance on every graph size. However, our experiments currently work with limited amount of data with respect to state-of-the-art approaches. Scaling our method to larger datasets requires more resources than we have available for this thesis, and presents a direct line of future work.
- We obtain competitive performance to the model in [8] with approximately 9 times reduction in the number of model parameters.
- We show the potential of extending the compositional heatmap construction to larger graph sampling sizes. That is, our model improves with the increase of subgraph sampling size at inference time, while the performance of the baseline gets worse.

Thesis Structure

The remainder of this thesis is structured as follows. In Chapter 2, we begin by laying the foundational concepts necessary for understanding our work. This is followed by an exploration of the existing literature and methodologies in Chapter 3. Chapter 4 describes the Neural Combinatorial Optimization (NCO) pipeline [4] that encompasses current approaches in the literature. In Chapter 5, we present the SizeShiftReg [9] method that we later adapt to our task. Chapter 6 motivates and describes our technical contributions. We then move to Chapter 7, where we describe our experiments followed by the interpretation of the results. Finally, in Chapter 8, we conclude by summarizing our findings, discussing the implications of our work, and suggesting potential directions for future research.

2

Preliminaries

2.1 Graph Neural Networks

Standard ML approaches expect vector data as input. However, in many domains, the most natural data representation has a graph-like structure. Modelling the relationships between the data can be even more important than the information stored in single data points. The typical example is the application in social networks where we can represent the users as data points, and the connections between them as edges. For this reason, we need ML architectures that are able to process graph data and extract valuable information both from the instances and the connections between them [10]. In this section, we provide an overview of Graph Neural Networks (GNNs) that were designed specifically for such scenarios.

2.1.1 Definitions

Graph

Firstly, we need to define what do we mean by "graph-like data". Formally, a graph $G = (V, E)$ is defined by a set of nodes V and a set of edges E between these nodes [11]. Graphs can be classified into two types based on the nature of their edges: directed and undirected graphs. In a directed graph, the edges have a direction, meaning that an edge (u, v) is distinct from an edge (v, u). This implies that the relationship from u to v is not necessarily the same as from v to u.

In contrast, an undirected graph contains edges that do not have a direction, so an edge (u, v) is identical to an edge (v, u). This reflects a symmetric relationship between nodes u and v. Generally, graphs can have multiple edges between each pair of nodes as well as self-edges, i.e., edges of type (u, u). However, in many cases, we restrict ourselves to simple undirected graphs, i.e., undirected graphs with at most one edge between each pair of nodes and no self-edges [11].

We often represent the graph as an adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{|V|\times|V|}$ where an entry A_{ij} is 1 if there is an edge between node i and node j, and 0 otherwise. Additionally, we can have weighted graphs where entries in the adjacency matrix represent the weight of and edge and can take any real value. For example, in a TSP instance, edge weights represent the distance between the nodes.

Node embeddings

Our goal is to find a projection of the nodes in a latent space (e.g., $\mathbb{R}^d$), where the geometric relations in the latent space correspond to the relationships created by the edges in the original graph [11, 12]. In this way, we can obtain a vectorial representation of a node that contains information about the single data point represented by it along with the information from its neighbors (the nodes connected by an edge). For a detailed overview of node embedding methods, please refer to [11].

Graph embeddings

Similarly to node embeddings, the goal of graph embedding techniques is to find a latent space representation of the graph as a whole. A typical way of achieving this is to first obtain the node embeddings for all the nodes of a graph and then combine them to form a representation of the entire graph. This can be done in different ways, e.g., summing the node representations, taking their mean, or even learning the mapping from node to graph embeddings.

Permutation invariance and equivariance

Consider a function f that takes an adjacency matrix $\boldsymbol{A}$ as input. The function f is *Permutation Invariant* if the following holds:

$$f(\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^T) = f(\boldsymbol{A}), \tag{2.1}$$

where $\boldsymbol{P}$ is a permutation matrix. On the other hand, the function f is said to be *Permutation Equivariant* if the following is true:

$$f(\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^T) = \boldsymbol{P}f(\boldsymbol{A}). \tag{2.2}$$

Informally, permutation invariance ensures that the output of a function will be the same independently of the order of rows and columns in the the adjacency matrix while permutation equivariant functions have an output that is consistent with the permutation of the adjacency matrix [11]. These concepts are crucial for making sure that the node and graph embeddings that we obtain are independent of the node ordering.

2.1.2 Neural Message Passing

In this section, we describe the *Neural Message Passing* framework along with the basic GNN architecture. Figure 2.1 illustrates the neural message passing framework for a single node. The image is taken from [11] that we follow as a reference for this section.

General Framework

The main idea of neural message passing is to exchange and update vector messages between nodes using neural networks. The embedding of a node in layer $(\ell+1)$ gathers the information from its own representation and representations of the neighboring nodes at the previous layer ℓ. We use the previously established notation along with denoting the matrix containing the node features with $\boldsymbol{X} \in \mathbb{R}^{d\times|V|}$, $\mathcal{N}(u)$ for the neighborhood of node u, and $\boldsymbol{h}_u^{(\ell)}$ to denote the embedding of node u at layer ℓ. The node embeddings are updated in the following way:

$$\begin{aligned} \boldsymbol{h}_u^{(\ell+1)} &= \text{UPDATE}^{(\ell)}\left(\boldsymbol{h}_u^{(\ell)}, \text{AGGREGATE}^{(\ell)}\left(\{\boldsymbol{h}_v^{(\ell)} : \forall v \in \mathcal{N}(u)\}\right)\right) \\ &= \text{UPDATE}^{(\ell)}\left(\boldsymbol{h}_u^{(\ell)}, \boldsymbol{m}_{\mathcal{N}(u)}^{(\ell)}\right), \end{aligned} \tag{2.3}$$

where UPDATE and AGGREGATE represent arbitrary differentiable functions [11]. In case of GNNs, these functions are neural networks with learnable parameters. The aforementioned message, obtained from the neighborhood of node u, is denoted as $\boldsymbol{m}_{\mathcal{N}(u)}^{(\ell)}$.

The inputs of the first layer are initial node features, i.e., $\boldsymbol{h}_u^{(0)} = \boldsymbol{x}_u, \forall u \in V$. Then, the node embeddings of each layer are updated by the formula written above for L iterations. The final embeddings for each node $\boldsymbol{h}_u^{(L)}$ can then be used for various purposes (for the downstream task, to obtain graph embeddings, etc.). It is also important to note that, as the AGGREGATE function takes a set as input, this definition of a GNN is permutation equivariant by design [11].

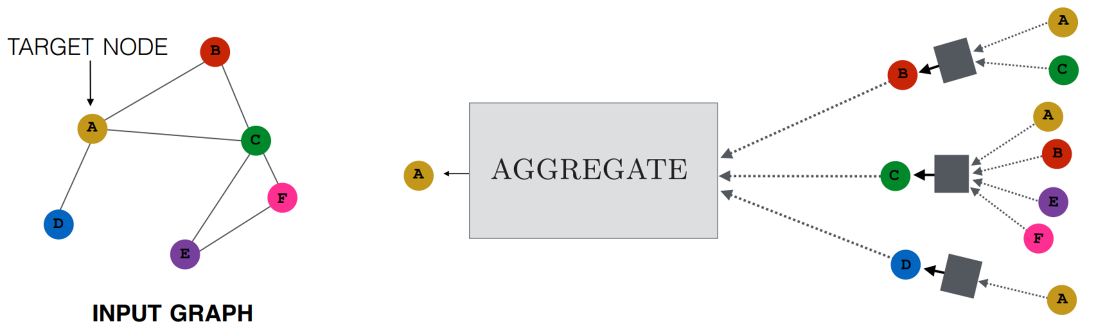


Figure 2.1: Overview of the neural message passing framework, focusing on the target node **A**. The messages are aggregated from **A**'s neighbors whose messages are based on information aggregated from their own neighbors. This process is repeated for a number of layers L. Image taken from [11].

.

The Basic GNN

In the general definition of the neural message passing framework, we can implement the GNN in different ways based on the choice of AGGREGATE and UPDATE functions. In this subsection, we describe the basic GNN that lays the foundations for most of the modern architectures. It is defined as follows:

$$\boldsymbol{h}_u^{(\ell)} = \sigma\left(\boldsymbol{W}_{\text{self}}^{(\ell)} \boldsymbol{h}_u^{(\ell-1)} + \boldsymbol{W}_{\text{neigh}}^{(\ell)} \sum_{v \in \mathcal{N}(u)} \boldsymbol{h}_v^{(\ell-1)} + \boldsymbol{b}^{(\ell)}\right), \tag{2.4}$$

where $\boldsymbol{W}_{\text{self}}^{(\ell)}, \boldsymbol{W}_{\text{neigh}}^{(\ell)} \in \mathbb{R}^{d^{(\ell)} \times d^{(\ell-1)}}$ denote trainable parameters, and σ is an elementwise non-linearity (e.g., ReLU). We often omit the bias term $\boldsymbol{b}^{(\ell)} \in \mathbb{R}^{d^{(\ell)}}$ to ease the notation.

2.1.3 Graph Convolutional Networks

There are multiple ways in which the Graph Convolutional Network (GCN) architecture can be motivated [13]. Here, we keep it simple and we present it as a basic GNN with the addition of symmetric normalization in the AGGREGATION function and self-loops in the UPDATE function [11]. GCN updates the hidden node embeddings as follows:

$$\boldsymbol{h}_u^{(\ell)} = \sigma\left(\boldsymbol{W}^{(\ell)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\boldsymbol{h}_v^{(\ell-1)}}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right). \tag{2.5}$$

Normalizing the embeddings by the size of the neighborhood can be a key ingredient in dealing with varying sizes of node degrees.

2.2 Combinatorial Optimization

2.2.1 Combinatorial Optimization Problems

Combinatorial Optimization (CO) problems consist of finding an optimal solution from a discrete (and finite) set of feasible solutions. We aim to pick a solution that has the maximum (minimum) value of the objective (cost) function that we have defined. The following is a formal definition of a combinatorial optimization problem as defined in [1].

An instance of a combinatorial optimization problem is a tuple (Ω, F, w), where Ω is a finite set, $F \subseteq 2^{\Omega}$ is the set of feasible solutions, and $c : 2^{\Omega} \to \mathbb{R}$ is a cost function with $c(S) = \sum_{\omega \in S} c(\omega)$ for $S \in F$. The goal is to select an element S^* (the optimal solution) that minimizes c over the feasible set F.

In this definition, we decide to frame it as a minimization problem, but it can equivalently be defined as the maximization problem. We also want to consider the decision version of the CO problems, where we ask if there exists an element in F with the cost smaller than a given value. The decision versions of the problems are important because they allow us to classify the problems as NP-complete [1].

2.2.2 Travelling Salesperson Problem

The Travelling Salesperson Problem (TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" For a more formal definition, we should first define a hamiltonian cycle. A hamiltonian cycle of an undirected graph $G = (V, E)$ is a simple cycle (no repeating nodes) that contains every node in V. TSP consists in finding a hamiltonian cycle of minimum weight [14]. Figure 2.2, taken from [1], illustrates an instance of the TSP and its optimal solution.

Many CO problems, including the TSP, that we find interesting belong to the NP-hard class, while their decision versions are NP-complete [1, 15]. Even though the worst-case scenario is intractable, many practical instances can be solved by defining them as linear optimization problems, constrained problems, or satisfiability problems, and using the methods designed for solving them. We refer the reader to [1] for a more comprehensive description of these techniques.

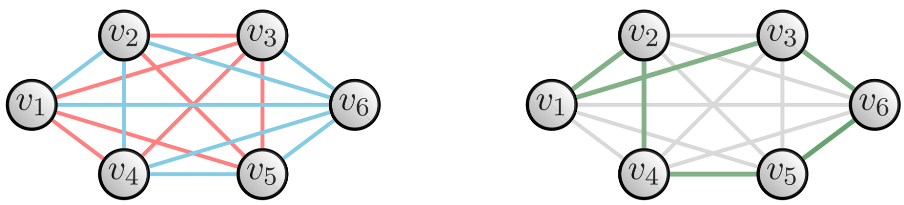


Figure 2.2: On the left, we can see a complete graph where blue edges have a weight of 1 and red edges have a weight of 2. On the right, we can see the optimal TSP solution denoted by the the green edges. Image taken from [1].

2.2.3 Traditional techniques for solving CO problems

There are various ways of categorizing techniques for solving CO problems. We follow the simple categorization of [1] where we distinguish between exact methods, local search and metaheuristics, and approximation algorithms.

Exact methods

As the name suggests, exact methods provide provably optimal solutions for CO problems. These include techniques like Integer Linear Programming solved through the branch-and-bound algorithm, which iteratively refines solutions using linear programming relaxations and cutting planes. If the CO problem is not directly modeled by integer programming, variations of the branch-and-bound method are applied using problem-specific relaxations. Additionally, Constraint Programming (CP) is another exact approach that systematically enumerates all possible solutions, leveraging constraints to reduce the search space and ensure optimality. The main downside of such approaches is the time required to obtain a solution that makes them inapplicable for large-scale instances.

Local search and metaheuristics

Local search and metaheuristics offer an alternative for handling large CO instances where finding the exact solution is infeasible. Local search begins with an initial candidate solution and iteratively improves it by exploring neighboring solutions. One of the main problems of this approach is the tendency to get stuck in local optima. However, it can be enhanced with metaheuristic techniques such as simulated annealing, tabu search, and genetic algorithms, which help navigate complex solution landscapes and mitigate the local optima problem.

Approximation algorithms

Approximation algorithms, typically running in polynomial time, provide solutions that are guaranteed to be within a certain bound of the optimal solution. Examples include Polynomial-Time Approximation Schemes (PTAS) and Fully Polynomial-Time Approximation Schemes (FPTAS), which offer solutions within a factor of $1 + \epsilon$ of the optimal, where ϵ is a tunable parameter that balances the trade-off between running time and solution quality. These provable guarantees of solution closeness to the optimum are the main difference with respect to local search.

2.3 Generalization

One of the most fundamental problems in ML is designing algorithms that will have good performance on inputs other than its training data [16]. For theoretical insights on the generalization in deep learning, please refer to [17, 18]. In this section, we first explain what we mean by generalization in ML followed by a quick overview of regularization techniques. Finally, we introduce the notion of *zero-shot learning* to lay the foundation for motivating our work.

2.3.1 Overfitting and Underfitting

Typically, when we train an ML model, the assumption is that we have access to the training set and the test set. Machine learning is different from optimization because we also want to minimize the generalization error (test error) in addition to minimizing the training error. We can define the generalization error as the expected value of the error on a new input [16]. As we do not have access to all the possible inputs, we estimate the generalization error by computing the error on the test set. In order to better understand how to tackle the challenge of finding this trade-off, we define the notions of *capacity*, *overfitting* and *underfitting*.

Underfitting occurs when a model is not able to properly fit the training set, i.e., the training error value is not sufficiently low. On the other hand, overfitting indicates that our model has learned the training set "too well", i.e., we have a large gap between the training error and the test error [16]. Whether a model overfits or underfits can be somewhat controlled by its capacity. The model's capacity determines its ability to fit various functions. Obviously, more complex models are more likely to overfit the training data while less complex ones are in danger of underfitting. Figure 2.3 provides an illustration of this trade-off. In addition to controlling the size of the set of functions that our model can learn, we can also control the identity of those functions. We introduce such techniques in the next section.

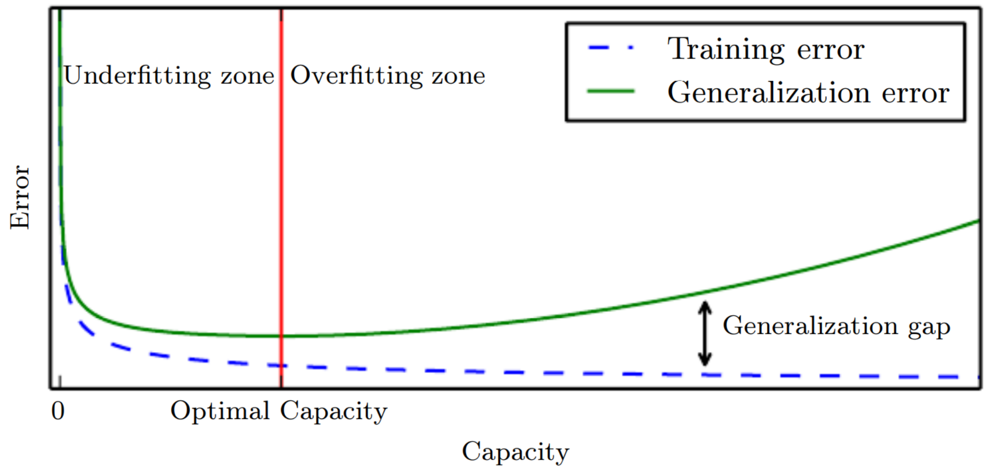


Figure 2.3: Typical relationship between model capacity and error rates. Initially, both training and generalization errors are high, indicating underfitting. As capacity increases, the training error decreases, but the gap between training and generalization errors gets bigger. Beyond a certain point, this gap surpasses the benefits of reduced training error, leading to overfitting, where the model capacity exceeds the optimal level. Image taken from [16].

2.3.2 Regularization

Regularization technique is broadly defined as any strategy that is aimed to improve the generalization error without improving the training error [16]. The aim is to prevent overfitting and allow the network to learn the underlying patterns that will be present in the unseen data. Usually, regularization methods come in the form of certain constraints on the model. A typical way to introduce the constraints is adding an additional term to the loss function $\mathcal{L}_{\text{tot}}$ in the following way:

$$\mathcal{L}_{\text{tot}} = \mathcal{L}_{\text{orig}} + \lambda \mathcal{L}_{\text{reg}}, \tag{2.6}$$

where $\mathcal{L}_{\text{orig}}$ is the original loss function (e.g., cross-entropy loss), $\mathcal{L}_{\text{reg}}$ is the regularization term, and λ is a hyperparameter that controls the trade-off between the two. We omit the model parameters θ to ease the notation.

Depending on how we define the additional term $\mathcal{L}_{\text{reg}}$, we can introduce different effects to our model. The most basic ones add constraints on the model's parameter values in order to simplify the model and avoid learning the noise present in the training data. The most common approaches of this kind are L_1 and L_2 parameter regularization, where $\mathcal{L}_{\text{reg}}$ is defined as the L_1 and L_2 norm of the model's weights, respectively. However, apart from adding a term that simply controls the magnitude of the weights, we can enforce various additional properties into our model. For example, we can guide the learning process to incorporate certain priors that we want to enforce. Moreover, we can even add the terms that can act as soft constraints to ensure the solution correctness.

Finally, we want to point out that adding an additional term into the loss function is not the only way of regularizing the model. In fact, we can use the techniques such as: data augmentation, multi-task learning, early stopping, parameter sharing, ensemble methods, dropout, adversarial training, and many more. For a detailed explanation of different regularization approaches, please refer to [16].

2.3.3 Zero-shot learning

In this section, we first introduce a general notion of zero-shot learning. Subsequently, we put it in the particular context for CO problems that we followed in our work.

In practical ML applications, we often have to work with limited amounts of data. Specifically, collecting labelled samples for all the categories can be very expensive. For this reason, a paradigm named *Few-shot learning* (a subset of the more general *n-shot learning*) has been proposed. Few-shot learning deals with machine learning problems where our dataset contains a limited amount of examples with supervised information [19]. It typically uses transfer learning and meta-learning-based approaches to efficiently adapt the model to classes with only a handful of labeled samples.

A more extreme case is zero-shot learning that aims to recognize and categorize objects whose instances may not have been seen during training [20]. So, our test set contains classes that are not present in the training set. To recognize unseen classes, the model focuses on auxiliary information that contains properties of both seen and unseen classes. An illustrative example would be training a model on domestic animals, and using the learned descriptive attributes to identify a new, unseen, wild animal. In the zero-shot learning setting, we test only on the instances of unseen categories. However, in order to have a model applicable in practice, we want to test both seen and unseen examples. This paradigm is called *generalized zero-shot learning*. For a detailed survey of these approaches, please refer to [21, 19, 20].

In our work, we are interested in a different form of zero-shot learning. Instead of focusing on classifying unseen categories, we aim to generalize solutions from smaller to larger instances without additional training. This setting is relevant for computationally-intensive CO problems where training on large sizes requires a lot of time and resources. For example, the authors of [4] demonstrate how learning the TSP can be unfeasible for large graphs. Some more recent works, such as [6], have been able to train on larger sizes directly, but large and complex models have many disadvantages. Developing models with the ability to learn the policy that scales across sizes is a fundamental problem for ML approaches in CO. For an overview of recent progress in this area, please refer to Chapter 3.

3
Related Work

Machine learning approaches for solving combinatorial optimization problems have become very popular in recent years. The main motivation is in discovering and leveraging common patterns in the given problem instances [2]. The authors of [1] provide a comprehensive overview of GNNs applications in the combinatorial optimization domain. They identify that distinguishing structural patterns in the data while scaling to large instances is one of the key challenges of this approach [1]. They provide a survey of the approaches focused on finding high-quality feasible solutions as well as works focused on proving the optimality of solutions. Finally, they discuss the recent neural algorithmic reasoning approach [22] to simulate the algorithmic behavior in GNNs [1]. The remainder of this section describes the approaches for finding high-quality feasible solutions for the TSP and Vehicle Routing Problems (VRPs) with special attention to their size-generalization ability.

GNN-based approaches for the TSP can be categorized based on the learning paradigm that they use, i.e., Supervised Learning (SL) [8, 23, 24, 6, 25], Reinforcement Learning (RL) [26, 27, 7, 28, 29, 30], and Unsupervised Learning (UL) [31, 32]. These methods perform well on the small instances, but their performance significantly downgrades on large-scale problems [24, 4]. SL methods require high-quality solutions that are expensive to obtain for the large-scale instances while the RL-based methods suffer from the sparse reward problem and device memory limitations [24, 33]. UL approach has been proposed to overcome the main downsides of the SL and RL approaches [31]. However, its ability to generalize to very large instances still has not been properly explored [32]. Various approaches applied in the context of the mentioned learning paradigms, that will be discussed in the following paragraphs, have mitigated the generalization problem with varying degree of success.

Another way to categorize ML solvers is into constructive and improvement methods [30]. Within the constructive approaches, as explained and analyzed by [4], there are autoregressive (AR) solvers [26, 27, 28, 24] and non-autoregressive (NAR) solvers [23, 8, 7, 25]. Autoregressive approach introduces the sequential inductive bias present in the TSP which improves size generalization, but its complex decoding scheme requires longer computation time. In contrast, NAR approaches produce heatmaps by predicting the probability of each edge belonging to the optimal solution independently. This leads to significantly faster inference compared to AR models, but their performance scales poorly to larger sizes than present in the training dataset [4]. Improvement solvers [34, 35, 36], on the other hand, start from an initial solution and learn to improve it in an iterative manner. This approach leads to better zero-shot generalization to large instances since it is typically used to guide decisions within classical search algorithms. However, it usually does not provide benefits in terms of execution time and it requires hand-crafted heuristics which may not be available for unexplored problems [4, 29]. Recently, diffusion based approaches [6, 30] have been proposed. They can be put in the existing categories, e.g., DIFUSCO [6] can be regarded as the constructive non-autoregressive approach, but it is also important to stress the novelty of applying diffusion to CO problems.

TSP, and VRPs in general, have shown to be more challenging in terms of scaling learning-based approaches to larger instances compared to non-sequential problems such as SAT [4]. The authors of [4] describe a unified pipeline that encompasses various constructive approaches. They demonstrate how directly learning on large instances is time-consuming while the zero-shot generalization from smaller to larger instances is challenging, and it often requires additional techniques [4]. Their analysis focuses on AR approaches because of the meaningful inductive bias that we mentioned earlier. Many subsequent works try tackling the challenge presented it this paper either by enabling more efficient training on larger sizes or aiming to improve the generalization performance in a zero-shot manner. We present an overview of both families of methods in the following paragraphs.

The aforementioned diffusion-based approaches do not focus on the zero-shot generalization, but they directly train on large instances [6, 30]. DIFUSCO [6] does require much more hardware for training compared to zero-shot generalization approaches, but it achieves good results with strong generalization across different graph sizes. On the other hand, the UL framework [31, 32, 37] manages to train directly on large instances while using only $\sim 0.2\%$ of unlabelled training samples and $\sim 10\%$ of model parameters with respect to SL and RL paradigms. This approach seems very promising, but its generalization ability requires more extensive exploration [32]. DIMES [7] is a novel RL approach that enables training on large instances by introducing a compact continuous space for parametrizing the underlying distribution of candidate solutions. DIMES achieves strong results, but it has longer inference time compared to some of the top-performing methods [29, 7].

On the other hand, the authors of [8] focus on the zero-shot generalization of NAR models. They do not improve the model's zero-shot generalization performance directly, but they opt for the divide-and-conquer approach. That is, they train a model on small, fixed-size instances and build a heatmap for the large instance using graph sampling, graph converting, and heatmaps merging. They achieve good performance on graphs up to 10000 nodes, but the method might not be suitable for real-time applications because it uses both the GNN with large number of parameters and an expensive MCTS on the produced heatmaps. The divide-and-conquer approach has also been successfully applied in more recent works [29, 28]. As opposed to the NAR architecture in [8], the authors of [28] employ an autoregressive approach to learn two levels of policies. The upper-level policy chooses a subset of nodes as the subproblem while the lower-level policy solves the open-loop TSP on the chosen subproblem. The proposed framework does not perform as well as the state-of-the-art approaches, but it achieves reasonable solution quality very quickly which makes it suitable for real-time applications [28]. Conversely, the authors of [29] use a divide-and-conquer approach by locally optimizing subparts of a solution. Starting with an initial solution, they sample a number of subproblems, feed them into the neural networks, and optimize only the subproblem that yields the largest improvement [29]. Their method is the first one to be tested on TSP instances of size 20000. Whereas the approaches discussed focus on breaking down the problem into smaller instances and combining the solutions, the method in [24] improves the generalization to larger instances by changing the model architecture. Differently from most of the constructive methods that typically use the heavy encoder and light decoder as initially proposed in [26], the authors of [24] introduce an architecture that puts most of the parameters in the decoder. They attempt to enforce the network to learn scale-independent features by refining the the learned node relationships during the training as opposed to one-shot node embedding learning in previous approaches. However, the approach has only been tested on instances up to 1000 nodes while the recent approaches usually test on graphs of sizes up to 10000. While the current approaches improve zero-shot generalization by either developing models that generalize to larger instances directly or by breaking down the problem into subproblems, they rarely focus on combining the two seemingly opposing alternatives. Moreover, direct generalization of AR approaches seems to be more extensively explored compared to its NAR counterpart due to the favorable sequential inductive bias.

4

Neural Combinatorial Optimization

In this section, we describe a generic pipeline for solving the TSP by constructive, learning-based approaches. The authors of [4] have defined this pipeline by unifying recent ideas in the literature. Firstly, we describe the pipeline skeleton followed by the various implementation choices for the particular stages. Figure 4.1 overviews the pipeline stages while Figure 4.2 illustrates each stage separately. Both images are taken from [4].

4.1 Neural Combinatorial Optimization Pipeline

Figure 4.1: Sequential stages of the NCO pipeline. Image taken from [4].

4.1.1 Problem Definition

In our work, we deal with 2D Euclidean TSP that the authors of [4] define as: "Given a set of nodes and the distances between each pair of nodes, what is the shortest possible route that visits each node and returns to the origin node?" Formally, we are given a fully-connected graph with n nodes in the two dimensional unit square $S = \{x_i\}_{i=1}^{n}$, where each $x_i \in [0, 1]^2$. The goal is to find a permutation of nodes $\boldsymbol{\pi} = (\pi_1, \pi_2, \dots \pi_n)$, named a tour, that visits each node exactly once and has the minimum total length. The total tour length $L(\boldsymbol{\pi}|S)$ is defined as:

$$L(\boldsymbol{\pi}|S) = \|x_{\pi_n} - x_{\pi_1}\|_2 + \sum_{i=1}^{n-1} \|x_{\pi_i} - x_{\pi_{i+1}}\|_2,$$

where $\|\cdot\|_2$ denotes the L_2 norm and π_i denotes the i-th node in the tour.

Graph Sparsification

We provide a typical definition of the TSP where input is a fully-connected graph. However, we will use *graph sparsification* to reduce the input sizes. Graph sparsification is a technique that reduces the number of edges in a graph while still maintaining its essential properties. In the literature, it is often used for efficiency when processing large scale instances [38, 4]. Regarding our application, it is also important for redefining the input to allow for effective use of SizeShiftReg [9]. We explain this in Section 6.2.2

There are various alternatives when choosing the specific graph sparsification technique. We opt for the one that showed the best generalization performance in [4]. That is, instead of a full graph, we connect each node to its $n \times k$%-nearest neighbors, where k is a parameter that controls the percentage of edges that we keep (e.g., $k = 0.5$ indicates that we keep half of the edges). In this way, the graphs have a fixed diameter across sizes which leads to the same number of message passing iterations necessary to spread the information, independently of the graph size [4]. In theory, this could remove the contribution of edges belonging to the optimal solution. However, in practice, it improves the performance for reasonable values of k, i.e., 0.2% or larger in [4], but some works remove significantly more edges [6].

4.1.2 Graph Embedding

We start by using GNN encoders that take the sparsified graph as input and produce embeddings for nodes, edges, and the whole input graph G. These embeddings will then be used as inputs to the subsequent step of the pipeline. Introduction to GNN architectures is provided in section 2.1. Detailed description of the particular architectures used in this work is presented in Section 4.2.

4.1.3 Solution Decoding

The Solution Decoding step takes the embeddings produced by the GNN and outputs the probabilities of nodes and/or edges for belonging to the optimal solution. The decoding can be categorized into Non-autoregressive Decoding (NAR) and Autoregressive Decoding (AR) [4].

The NAR approach performs a link-prediction task on the embeddings produced by the GNN. Each edge is assigned a probability of belonging to the optimal solution independently of all the other edges. The simplest and the most-common way of defining this predictor is a simple Multi-Layer Perceptron (MLP) on the edge, node, and/or graph representations [23, 8, 4]. Section 4.3 provides a description of the edge-classifiers used in this work.

NAR decoding, though fast, does not introduce a sequential inductive bias that is natural for the VRPs [4]. For this reason, the solution decoding can also model this inductive bias by producing node probabilities dependent on each other. For example, the attention model designed in [26] starts from a random node and outputs a probability distribution over its neighbors at each step.

4.1.4 Solution Search

After obtaining the probabilities for edges or nodes of belonging to the optimal solution, we can perform various search techniques to transform them into discrete decisions. The implementation of the search technique will differ depending on the solution decoding (AR vs NAR). As our work uses a NAR approach, we focus on the solution search techniques designed for this solution decoding paradigm. Search techniques vary in complexity and performance. In [4], the authors compare the Greedy Search, Beam Search, and Solution Sampling. However, there are more sophisticated search techniques such as the one presented in [25], that is based on Dynamic Programming (DP). Moreover, the authors of [8] use MCTS, which is a form of reinforcement learning for solution optimization. A detailed comparison of solution search techniques is provided in Section 4.4.

4.1.5 Policy Learning

Policy Learning refers to the ML paradigm used to train the network. As mentioned in the related work section, all of the three main paradigms (SL, RL, and UL) have been applied to the TSP with varying degrees of success. We focus on the SL paradigm in our work. In case of NAR decoders, we minimize the binary cross-entropy loss between our model's edge predictions and the ground-truth TSP solution [23, 8, 4]. On the other hand, for AR architectures, we can train the network by minimizing the cross-entropy loss between the predicted probability distribution over all edges leaving the current node and the next node from the ground-truth tour [39, 4]. Teacher-forcing is used to stabilize the training [40].

4.2 Graph Embedding - Architectures

There are numerous possibilities for designing GNN encoder architectures for learning the TSP. In this section, we provide details about the two architectures that we use in our experiments. We inherit the concepts and notation from Section 2.1, where we introduce GNNs.

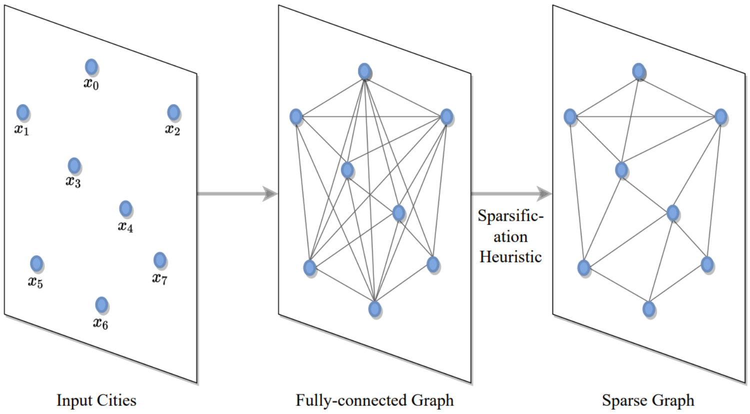
4.2.1 Anisotropic Graph Convolutional Network

Anisotropic Aggregation makes the aggregation function directional by scaling the contribution of the neighboring nodes' representations $\boldsymbol{h}_j, \forall j \in \mathcal{N}_i$ through edge gates $\sigma(\boldsymbol{e}_{ij})$ [41, 4]. We take the model from [4] presented by the following formulas:

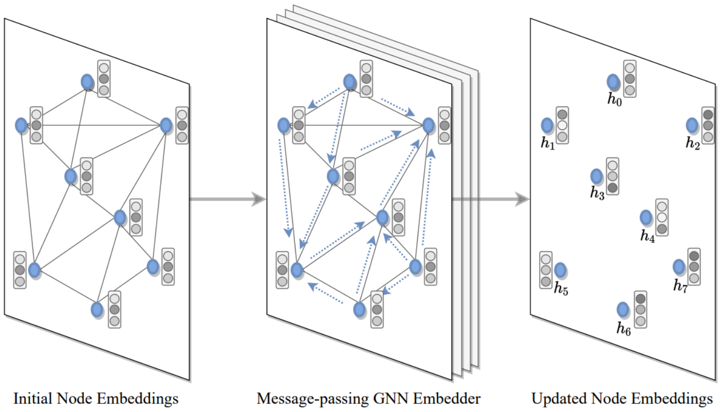
$$\boldsymbol{h}_i^{(\ell+1)} = \boldsymbol{h}_i^{(\ell)} + \mathrm{ReLU}\left(\mathrm{NORM}\left(\boldsymbol{U}^{(\ell)}\boldsymbol{h}_i^{(\ell)} + \mathrm{AGG}_{j\in\mathcal{N}_i}\left(\sigma(\boldsymbol{e}_{ij}^{(\ell)}) \odot \boldsymbol{V}^{(\ell)}\boldsymbol{h}_j^{(\ell)}\right)\right)\right), \quad (4.1)$$

$$\boldsymbol{e}_{ij}^{(\ell+1)} = \boldsymbol{e}_{ij}^{(\ell)} + \mathrm{ReLU}\left(\mathrm{NORM}\left(\boldsymbol{A}^{(\ell)}\boldsymbol{e}_{ij}^{(\ell)} + \boldsymbol{B}^{(\ell)}\boldsymbol{h}_i^{(\ell)} + \boldsymbol{C}^{(\ell)}\boldsymbol{h}_j^{(\ell)}\right)\right), \quad (4.2)$$

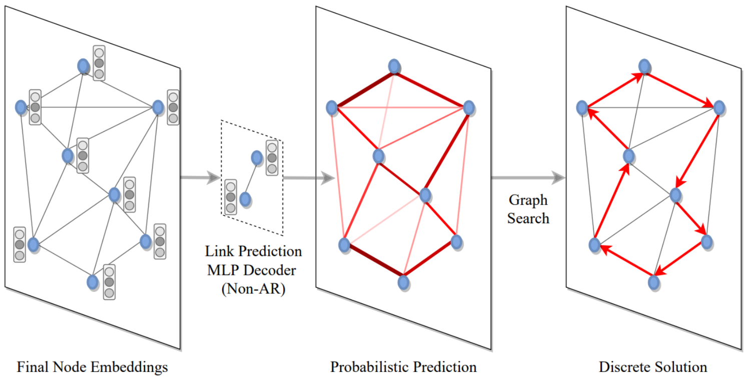
where $\boldsymbol{U}^{(\ell)}, \boldsymbol{V}^{(\ell)}, \boldsymbol{A}^{(\ell)}, \boldsymbol{B}^{(\ell)}, \boldsymbol{C}^{(\ell)}$ are learnable parameters, NORM denotes the normalization layer that can incorporate BatchNorm [42] or LayerNorm [43], σ is the sigmoid function, $\odot$ is the Hadamard product, ReLU denotes the rectified linear unit [44], and AGG is the neighborhood aggregation function (abbreviated from AGGREGATE in Section 2.1). Network inputs $\boldsymbol{h}_i^{(0)}, \boldsymbol{e}_{ij}^{(0)}$ are d-dimensional linear projections of the node coordinates x_i and the Euclidean distance $\|x_i - x_j\|_2$, respectively [4].



(a) Problem Definition



(b) Graph Embedding



(c) Solution Decoding and Solution Search

Figure 4.2: Depiction of NCO pipeline stages. Model is trained end-to-end via SL, RL, or UL (Policy Learning stage). Image taken from [4].

4.2.2 Att-GCRN

The graph convolutional residual neural network with attention mechanism (Att-GCRN) architecture is originally used by [23], and subsequently adapted in [8]. Firstly, the inputs $\boldsymbol{x}_i$ and d_{ij}, representing the coordinates of node i (represented as a 2×1 vector) and the distance of edge (i, j), are embedded into a d-dimensional vector space:

$$\boldsymbol{v}_i^{(0)} = \boldsymbol{W}_1 \boldsymbol{x}_i + \boldsymbol{a}_i, \tag{4.3}$$

$$\boldsymbol{e}_{ij}^{(0)} = \boldsymbol{W}_2 d_{ij} + \boldsymbol{b}_{ij}, \tag{4.4}$$

where $\boldsymbol{v}_i^{(0)}$ and $\boldsymbol{e}_{ij}^{(0)}$ denote the d-dimensional embeddings corresponding to node i and edge (i, j). $\boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{a}_i$, and $\boldsymbol{b}_{ij}$ are trainable parameters. We obtain a $d \times n$ matrix $\boldsymbol{V}^{(0)}$ representing the embedding of all the vertices, while a tensor $\boldsymbol{E}^{(0)}$ of dimension $d \times n \times n$ contains the embeddings of all the edges [8].

The obtained embeddings are now fed in to the feature extractor of L layers. The elements $\boldsymbol{v}^{(\ell)}$ and $\boldsymbol{e}^{(\ell)}$, denoting the node and edge representations of the ℓ-th layer, respectively, are calculated as follows:

$$\boldsymbol{v}_i^{(\ell)} = \boldsymbol{v}_i^{(\ell-1)} + \mathrm{ReLU}\left(\mathrm{BN}\left(\mathrm{Conv}(\boldsymbol{v}_i^{(\ell-1)}) + \sum_{j \neq i} \alpha_{ij}^{(\ell)} \odot \mathrm{Conv}(\boldsymbol{v}_j^{(\ell-1)})\right)\right), \tag{4.5}$$

$$\boldsymbol{e}_{ij}^{(\ell)} = \boldsymbol{e}_{ij}^{(\ell-1)} + \mathrm{ReLU}\left(\mathrm{BN}\left(\mathrm{Conv}^*(\boldsymbol{e}_{ij}^{(\ell-1)}) + \mathrm{Conv}(\boldsymbol{v}_i^{(\ell-1)}) + \mathrm{Conv}(\boldsymbol{v}_j^{(\ell-1)})\right)\right), \tag{4.6}$$

where ReLU denotes rectified linear unit [44], BN denotes batch normalization [42], and Conv represents a one-dimensional convolutional layer with the stride of 1 and d kernels of size 1×1 [8]. Conv* differs from Conv by having d kernels of size $1 \times 1 \times 1$, but the output of both convolutional layers is a vector of dimension $d \times 1$. As this architecture uses the attention mechanism [45], we have $\alpha_{ij}^{(\ell)}$ denoting the attention coefficient of the edge (i, j):

$$\alpha_{ij}^{(\ell)} = \frac{\exp\left(\text{LeakyReLU}\left(\text{Conv}^{\diamond}(\boldsymbol{e}_{ij}^{(\ell)})\right)\right)}{\sum_{k\neq i}\exp\left(\text{LeakyReLU}\left(\text{Conv}^{\diamond}(\boldsymbol{e}_{ik}^{(\ell)})\right)\right)}, \tag{4.7}$$

where LeakyReLU denotes the leaky rectified linear unit [44]. $\text{Conv}^{\diamond}$ is a one-dimensional convolutional layer consisting of one $1 \times 1 \times 1$ kernel with stride 1, and it outputs a real value [8].

Edge representations at the final layer $\boldsymbol{e}^{(L)}$ are then fed into the classification network (solution decoding step from the pipeline described earlier). The classification network outputs a matrix of shape $n \times n$ representing the heatmap $\boldsymbol{P}$, where P_{ij} is the probability of the edge (i, j) belonging to the optimal solution. The concrete architecture used the for the edge classification is described in Section 4.3.

Finally, the Att-GCRN model [8] uses a focal loss function [46] to calculate the accumulated loss:

$$\mathcal{L}(\boldsymbol{P}) = -\frac{1}{M}\sum_{m=1}^{M}\sum_{i=1}^{N}\sum_{j\neq i}\left(\beta_0(1 - I_{ij}^{(m)}) + \beta_1 \cdot I_{ij}^{(m)}\right)\left(I_{ij}^{(m)} - P_{ij}^{(m)}\right)\log(P_{ij}^{(m)}), \tag{4.8}$$

where $\boldsymbol{I}^m$ is a $n \times n$ matrix corresponding to the m-th training instance. The element I_{ij}^m equals 1 if the edge (i, j) belongs to the ground-truth solution of the m-th training instance, and it is equal to 0 otherwise. P_{ij}^m is the probability assigned to an edge (i, j) corresponding to m-th training instance. β_0 and β_1 are balanced coefficients relative to the edges not belonging to (or belonging to) the ground-truth solutions, which are calculated as follows [47, 8]:

$$\beta_0 = \frac{n^2}{n^2 - 2n}, \quad \beta_1 = \frac{n^2}{2n}. \tag{4.9}$$

4.3 NAR Solution decoding - architectures

As described in Section 4.1.3, NAR solution decoding is defined as a link-prediction task on the embeddings produced by the GNN encoder. In general, the link-predictor can be of arbitrary complexity, but it is usually implemented using a simple MLP [8, 4, 23].

AGCN Solution Decoding

In particular, the authors of [4] implement a two-layer MLP on the node embeddings from the final GNN encoder layer L (the corresponding GNN encoder is described in 4.2.1):

$$\hat{P}_{ij} = \boldsymbol{W}_2 \left(\mathrm{ReLU} \left(\boldsymbol{W}_1 \left([\boldsymbol{h}_G, \boldsymbol{h}_i^{(L)}, \boldsymbol{h}_j^{(L)}] \right) \right) \right), \tag{4.10}$$

where $\boldsymbol{W}_1 \in \mathbb{R}^{3d \times d}$, $\boldsymbol{W}_2 \in \mathbb{R}^{d \times 2}$ are trainable parameters, $[\cdot, \cdot, \cdot]$ is the concatenation operator, and $\boldsymbol{h}_G$ is the graph embedding. It can be obtained in various ways (mean, max, sum), but in [4] mean has shown the best performance:

$$\boldsymbol{h}_G = \frac{1}{n} \sum_{i=0}^{n} \boldsymbol{h}_i^{(L)}. \tag{4.11}$$

The final probabilities of each edge P_{ij} are obtained by applying softmax to logits $\hat{P}_{ij}$.

Att-GCRN Solution Decoding

The classification network in [8] contains three MLP layers operating on the final encoder edge embeddings $\boldsymbol{e}^{(L)}$. The heatmap $\boldsymbol{P}$ is obtained as:

$$\boldsymbol{P} = \mathrm{MLP}(\boldsymbol{e}^{(L)}), \tag{4.12}$$

where MLP contains three fully connected layers.

4.4 Solution Search Techniques

As described in the NCO pipeline, we need to perform a search on the space of probabilities obtained by the model to construct the solution. In this subsection, we describe the solution search techniques used for NAR approach, i.e., those that work on probability heatmaps.

4.4.1 Greedy Search

Greedy search is the simplest and the fastest technique used to obtain the final tour prediction. We traverse a graph starting from a random node and mask already visited nodes to obtain a TSP tour. The probability to traverse from a node to its neighbor is based on the probability (output of our model) off the connecting edge. Hence, the probability of a partial tour $\boldsymbol{\pi}'$ is defined as $p(\boldsymbol{\pi}') = \prod_{j' \sim i' \in \boldsymbol{\pi}} P_{i'j'}$, where we traverse from node i to node j [4].

4.4.2 Beam Search

We can improve greedy search by keeping b most probable tours at each step instead of just one. This algorithm is called beam search, and the number of tours we keep at each step is the beam width. Increasing the beam width (or simply searching longer) can improve the solution quality, but it comes with a trade-off of using more time. Moreover, using a very large beam width can overshadow the model's poor generalization ability [4, 48].

4.4.3 Dynamic Programming Search

Deep Policy Dynamic Programming (DPDP) [25] is a form of beam search that, instead of searching the solution space, operates over the Dynamic Programming (DP) state space. It uses the heatmap predicted by the model to derive the scoring policy at each step of the search. DPDP is a general framework that requires definition of several ingredients depending on the specific problem. Please refer to [25] for details on these ingredients and how they should be defined in order to construct the TSP state space.

4.4.4 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a reinforcement learning-based approach that operates by sampling k-opt transformations guided by the generated probability heatmap $\boldsymbol{P}$. Heatmap-based approaches in the literature use variations of MCTS for finding high-quality solutions [6, 8, 31, 7]. In our experiments, we use the implementation from [8]. Firstly, we give an overview of the procedure in the next subsection followed by multiple subsections that break-down each one of its components.

Search overview

The search starts from an initial tour that is obtained using the heatmap probabilities. Then, it iteratively improves the solution by performing 2-opt moves until it finds a local optimum. Once there are no better solutions within the small neighborhood, the search expands to a larger neighborhood with k-opt moves, for $k > 2$. Given the infeasible size of the enlarged neighborhood, promising moves are sampled based on the heatmap obtained from the model. Moreover, the heatmap probabilities are updated using RL with the goal of reinforcing the edges that have been present in good solutions. After the sampling pool is exhausted, the search jumps to another state by repeating the initialization process from the beginning.

State and action representation

MCTS starts from an initial tour $\boldsymbol{\pi}$ and iteratively select an action a to reach an improved solution $\boldsymbol{\pi}^*$ until we run out of search time. Each solution is represented as a permutation $(\pi_1, \pi_2, \dots \pi_n)$ of all the vertices (which induces the edges between them) while an action can be defined as a k-opt transformation ($2 \leq k \leq n$), where we delete k edges and add k new edges to the solution. The problem with this formulation is that it can lead to an unfeasible solution as we are allowed to arbitrarily delete and add k edges. Hence, the authors of [8] use a different representation that requires only k (previous one needs $2k$) decisions for each action and guarantees the feasibility of the solution. Action is represented as $a = (a_1, b_1, a_2, b_2, \dots, a_k, b_k, a_{k+1})$, where $a_1 = a_{k+1}$. Now, we can represent an action as deleting k edges of the form (a_i, b_i) and adding k edges of the form (b_i, a_{i+1}) for $1 \leq i \leq k$. Using this representation, we reduce to k decisions because b_i can be uniquely determined based on the choice of a_i. Moreover, this action representation guarantees to transform the state to a cycle (a feasible solution). Please refer to [8] for a detailed explanation of these claims followed by an example. Given these notions, we can define the difference between the tour lengths obtained by applying the action a as:

$$\begin{aligned} \Delta(\boldsymbol{\pi}, \boldsymbol{\pi}^*) &= L(\boldsymbol{\pi}^*) - L(\boldsymbol{\pi}) \\ &= \sum_{i=1}^{k} d_{b_i, a_{i+1}} - \sum_{i=1}^{k} d_{a_i b_i}, \end{aligned} \tag{4.13}$$

where $L(\boldsymbol{\pi})$ denotes the length of the tour $\boldsymbol{\pi}$ and d_{ij} is the distance on the edge e_{ij}. Negative difference indicates that the tour $\boldsymbol{\pi}^*$ is better.

Initial state and promising actions

Before getting into the details of the search, we should clarify how we find an initial state as well as which actions we want to consider at each step. State initialization starts from a random node π_1 and iteratively selects node π_i from the set of unvisited nodes. Unvisited node j is selected with the probability $\exp(P_{\pi_i j})$ and the sum of the probabilities of the candidate vertices is equal to 1. Instead of considering all the actions with a certain k, we remove those that contain an edge (b_i, a_{i+1}) with $P_{b_i a_{i+1}} < 10^{-4}$. The remaining actions are called promising actions.

Small neighborhood search

The search begins in the small neighborhood of the current state by evaluating promising actions with $k = 2$ one by one. We do not necessarily consider all such actions because we move to the better solution as soon as it is found. We repeat this process until there no more promising actions with $k = 2$ that can improve our solution. The goal is to use this simple procedure to quickly reach the local optima.

Targeted Sampling within Enlarged Neighborhood

Finding a high-quality solution requires searching within a larger neighborhood, i.e., evaluating actions with $k > 2$. As this is very expensive in general, we sample a subset of promising actions guided by our heatmap $\boldsymbol{P}$ that will be updated throughout the iterations. The procedure is composed of four steps: (1) Initialization, (2) Simulation, (3) Selection, and (4) Backpropagation. Figure 4.3, taken from [8], illustrates this sequence of steps.

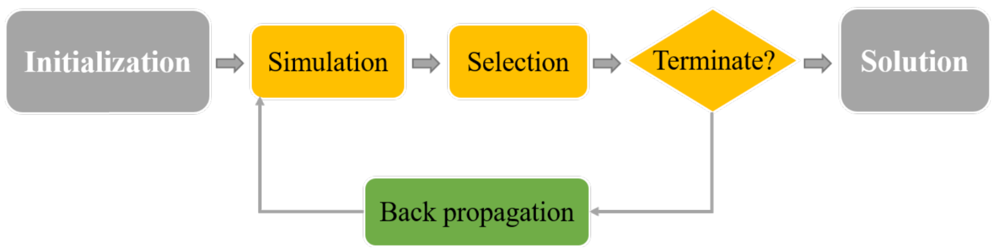


Figure 4.3: Overview of the targeted sampling within enlarged neighborhood. Image taken from [8].

Initialization

We define two symmetric $n \times n$ matrices, $\boldsymbol{W}$ and $\boldsymbol{Q}$. Elements of $\boldsymbol{W}$ control the probability of choosing vertex j after vertex i, and are initialized as $100 \times P_{ij}$. On the other hand, Q_{ij} keeps track of how many times the edge (i, j) is picked during simulations, and every value is initially 0. We also need a variable c to keep track of the total number of simulated actions.

Simulation

In the simulation process, we generate a number of actions to be considered from a state $\boldsymbol{\pi}$. We remind you that we make a series of decision choosing a_i for $1 \leq i \leq k$ while b_i can be determined uniquely once a_i is chosen. So, we estimate the potential $Z_{b_i j}$ of each edge $(b_i, j), j \neq b_i$, based on the following formula:

$$Z_{b_i j} = \frac{W_{b_i j}}{\Omega_{b_i}} + \alpha \sqrt{\frac{\ln(c+1)}{Q_{b_i j}+1}}, \tag{4.14}$$

where $\Omega_{b_i} = \frac{\sum_{j \neq b_i} W_{b_i j}}{\sum_{j \neq b_i} 1}$ is the averaged value in $\boldsymbol{W}$ of all the nodes relative to b_i. The potential $Z_{b_i j}$ balances exploration and exploitation with the term $\frac{W_{b_i j}}{\Omega_{b_i}}$ that prefers highly-probable edges and the term $\alpha \sqrt{\frac{\ln(c+1)}{Q_{b_i j}+1}}$ that prefers under-explored edges. The hyperparameter α allows for experimentation with levels of intensification and diversification.

After randomly choosing an initial a_1 (that implies b_1), we define the set V_p of all the nodes that have $W_{b_i j} \geq 1$ besides a_1 and the node already connected to b_i. We choose a node from V_p with probability:

$$P_j = \frac{Z_{b_i j}}{\sum_{p \in V_p} Z_{b_i p}}. \tag{4.15}$$

The only exception is when we want to close the loop, which happens if connecting b_i to a_1 is an improving action or $i \geq 10$. In two those cases, we let $a_{i+1} = a_1$. We generate actions until we find an improving action or until we reach a maximum number of actions controlled by the parameter h.

Selection

If the simulation process finds an action leading to a better solution, we move to a new state $\boldsymbol{\pi}^{new}$. Otherwise, we move out of the current search area by jumping to a random state using the initialization method described previously.

Backpropagation

We want to update the values of our matrices $\boldsymbol{W}$ and $\boldsymbol{Q}$ as well as the variable c throughout the iterations. The count c is simply incremented whenever an action is evaluated, while $Q_{b_i a_{i+1}}$ increases by 1 for each edge (b_i, a_{i+1}) in the evaluated action. Most importantly, whenever we reach an improved state $\boldsymbol{\pi}^{new}$ from a current state $\boldsymbol{\pi}$ by applying $a = (a_1, b_1, a_2, b_2, \ldots, a_k, b_k, a_{k+1})$, we update our probability matrix in the following way:

$$W_{b_i a_{i+1}} \leftarrow W_{b_i a_{i+1}} + \beta \left[\exp\left(\frac{L(\boldsymbol{\pi}) - L(\boldsymbol{\pi}^{new})}{L(\boldsymbol{\pi})} \right) - 1 \right], \tag{4.16}$$

where β is a hyperparameter used to control how much we increase the probability of edges appearing in the improved state. Backpropagation is used to make the sampling process more targeted by increasing the probability of favorable edges.

Termination

The process of simulation, selection, and backpropagation repeats until there are no more improving sampled actions. At that point, the search jumps to a new state where it again starts from a local neighborhood search. The whole procedure stops once the time controlled by the parameter t has elapsed. In the end, the best solution found within the allowed time is returned.

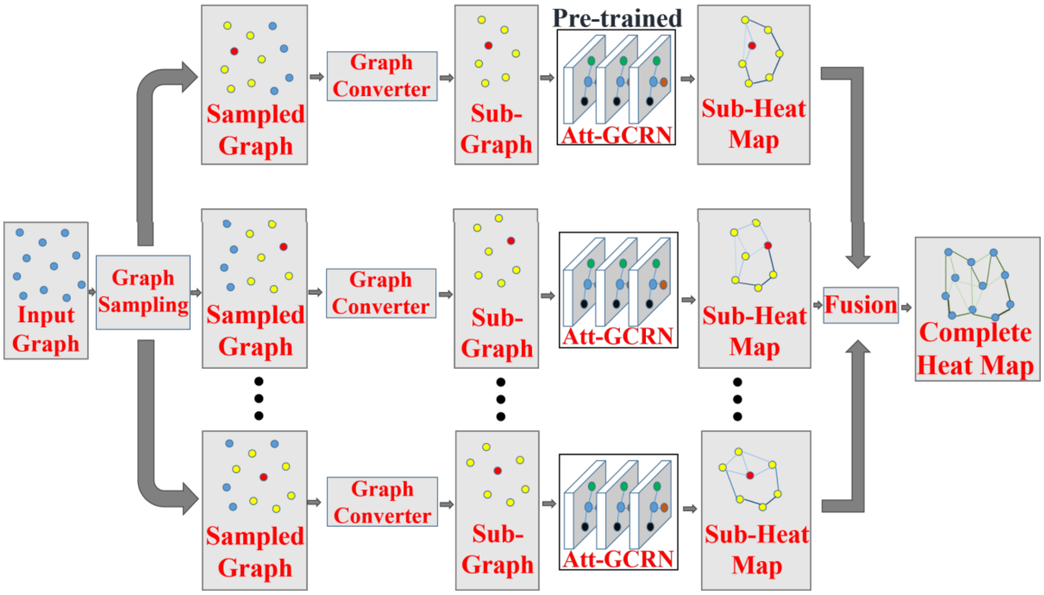


Figure 4.4: Overview of the method to produce compositional heatmaps. Image taken from [8].

4.5 Compositional Heatmaps

In [4], the authors identify two crucial challenges for learning-based TSP approaches:

- Training on large-scale instances is very computationally expensive.
- Current approaches scale poorly to larger instances. This is especially true for NAR architectures.

For this reason, various divide-and-conquer approaches have been recently proposed [8, 29, 28]. In this section, we present the compositional approach from [8] that we use throughout our work.

If you go back to steps 2 and 3 of our NCO pipeline, you can see that we use our model directly on the graphs at inference time. However, we can introduce additional steps into heatmap construction to have a compositional approach in cases where training and test graph sizes significantly differ.

We sample a number of subgraphs of size m (m is a hyperparameter) from the original graph and use our model to obtain the heatmaps for all the subgraphs independently. Then, we can merge the heatmaps from the subgraphs into one big heatmap for the large-scale graph. By setting m to the training graph size as in [8], we can expect the model to perform well on the sampled subgraphs. In this way, we hope to mitigate the generalization problem and obtain better heatmaps as the input to the search procedure. In Figure 4.4 , taken from [8], we provide an illustrative overview of the procedure. In the following paragraphs, we break down each step.

Graph Sampling

We use graph sampling to obtain a certain number of subgraphs from the original graph $G = (V, E)$. It starts by initializing counters o_i and O_{ij} to zero for each vertex $i \in V$ and edge $(i, j) \in E$. In each iteration, the vertex with the smallest o_i is chosen (ties are broken randomly) as the clustering center. The k-nearest neighbors algorithm (k is actually set to m in this case) is then used to form a subgraph G' containing m vertices. For every vertex i and edge (i, j) in G', the corresponding counters o_i and O_{ij} are incremented by one. This process continues until the minimum value of o_i reaches a lower bound ω (a hyperparameter).

Graph Converting

As, in practice, we synthetically generate our train data from the uniform distribution, the model will train on graphs with randomly distributed nodes in a unit square. Our subgraph sampling can change this distribution and potentially decrease the model performance. To ensure that the extracted subgraph G' has the same distribution, we convert it into a new graph G''. First, we calculate the minimal and maximal values of the horizontal and vertical coordinates for the vertices in G' and denote them as $x^{\text{min}}, x^{\text{max}}, y^{\text{min}}, y^{\text{max}}$. We then calculate the normalization factor $s = \frac{1}{\max(x^{\text{max}}-x^{\text{min}}, y^{\text{max}}-y^{\text{min}})}$), and normalize the coordinates (x_i, y_i) of every node $i \in G'$ as follows:

$$x_i^{\text{new}} \leftarrow s \times (x_i - x^{\text{min}}), \quad y_i^{\text{new}} \leftarrow s \times (y_i - y^{\text{min}}). \tag{4.17}$$

After applying the normalization, we have subgraphs with uniformly distributed nodes.

Merging Subgraph Heatmaps

Heatmaps for each of the z (a hyperparameter) subgraphs G'' are obtained as an output of our model. We then merge them into a complete heatmap for the original graph G. The probability

P_{ij} of each edge (i, j) being a part of the optimal solution is computed in the following way:

$$P_{ij} = \frac{1}{O_{ij}} \times \sum_{k=1}^{z} P''_{ij}(k),$$

where $P''_{ij}(k)$ represents the probability that edge (i, j) belongs to the optimal solution in the k-th subgraph G''. Edges with $P_{ij} < 10^{-4}$ are marked as unpromising and are excluded to simplify the search space.

5
SizeShiftReg

In this chapter, we describe SizeShiftReg, a regularization method introduced in [9]. We begin with the high-level view of the whole technique followed by the description of its components. Finally, we discuss an inherent limitation of the approach (and similar approaches) noted by the authors.

5.1 SizeShiftReg Overview

SizeShiftReg is a regularization method designed to improve size-generalization capabilities of Graph Neural Networks for the graph classification task. In many applications, graphs of varying sizes are encountered, and while GNNs theoretically handle graphs of any size, their performance often degrades when the size of the test graphs significantly differs from the training size. SizeShiftReg deals with this problem by simulating a size shift in the training data and regularizing the GNN to be robust to such a shift. Size shift is obtained using graph coarsening techniques while the regularization aims to minimize the discrepancy between the node embedding distributions of the original graph and its size-shifted versions. This method is model-agnostic and it does not assume access to any additional data apart from the training set. Experimental results in [9] show its effectiveness on the TUDataset library [49] with a train/test split specifically designed for size-generalization evaluation. Moreover, the results show the method's robustness to the choice of the model and the graph coarsening technique. In Figure 5.1, taken from the original paper, we can see an illustration of the whole pipeline.

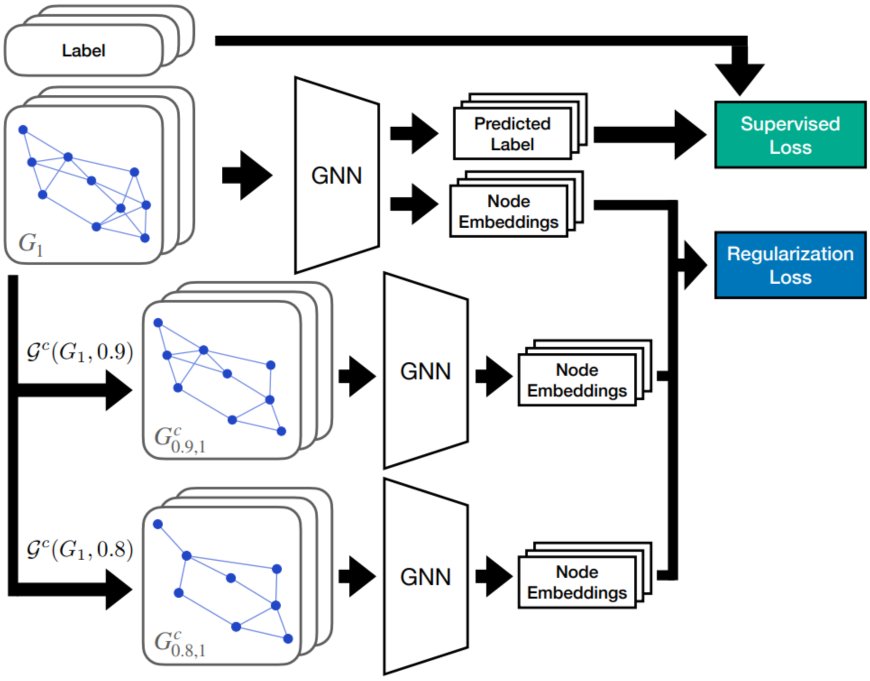


Figure 5.1: Overview of the SizeShiftReg pipeline. Image taken from [9].

5.2 Graph Coarsening

Graph coarsening is a technique used to reduce the size of a graph while preserving its essential structural properties. A typical application of graph coarsening is to reduce the computational load by working on smaller graphs with the main properties of their original versions [5]. On the other hand, in SizeShiftReg, graph coarsening creates smaller versions of the original graphs to simulate a size shift in the training data.

Formally, given a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges, a coarsened graph $G_c = (V_c, E_c)$ is obtained by partitioning V into subsets and merging the nodes within each subset into a single node. The edges in the coarsened graph are determined by the connectivity between these subsets. The goal is to generate a coarsened graph where the number of nodes is reduced (i.e., $|V_c| < |V|$) while maintaining key properties such as the spectral characteristics of the original graph. This is crucial for ensuring that the coarsened graph remains a valid representation of the original structure.

The authors of [9] have tested four different coarsening methods. The first two are specialized techniques introduced in [5], i.e., Spectral Graph Coarsening and Multi-level Graph Coarsening while the other two are baselines based on standard spectral clustering and k-means clustering algorithms. The specialized methods do provide the best results in general, but the overall technique maintains similar performance independently of the coarsening method.

5.3 Central Moment Discrepancy

The Central Moment Discrepancy (CMD) [50] is a statistical metric that measures the discrepancy between two high-dimensional distributions. In the context of SizeShiftReg, CMD is used to regularize the GNN by enforcing that the distribution of node embeddings generated for the original graph and its coarsened versions are similar. In order to use CMD on the node embeddings, we need to treat them as bounded high-dimensional distributions by normalizing the GNN layers and/or using bounded activation functions.

CMD is calculated by comparing the central moments (mean, variance, skewness, etc.) of the distributions up to a certain order. For two distributions p and q, the CMD between them is given by:

$$\mathrm{CMD}(p,q) = \frac{1}{|b-a|}\|\mathbb{E}(p) - \mathbb{E}(q)\|_2 + \sum_{k=2}^{\infty} \frac{1}{|b-a|^k}\|c_k(p) - c_k(q)\|_2, \tag{5.1}$$

where $\mathbb{E}(p)$ and $\mathbb{E}(q)$ are the expectations, interval $[a,b]^d$ is the support of probability distributions, and c_k represents their k-th order moment. In practice, the number of moments considered is usually limited to 5 to reduce the computational complexity.

5.4 Implementation Details

Now that we provided the high-level idea as well as the background knowledge for SizeShiftReg, we present concrete implementation steps as described in [9].

5.4.1 Coarsened Dataset

The first step is to simulate a shift by creating a coarsened version of the training dataset. Let $\mathcal{G}^c : (G, r) \to G^c_r$ be a coarsening function that returns a coarsened graph G^c_r with $\lfloor r \times n \rfloor$ nodes, where $r \in [0, 1]$. Given an original dataset of graphs $\mathcal{D} = \{G_1, G_2, \cdots G_l\}$ and a set of coarsening ratios $C = \{r_1, r_2, \ldots, r_k\}$, the original training graphs are coarsened multiple times, resulting in a set of coarsened datasets. We denote them by $\mathcal{D}_{r_j}$ for each ratio, i.e. $\mathcal{D}_{r_j} = \{G^c_{r_j,1} = \mathcal{G}^c(G_1, r_j), G^c_{r_j,2} = \mathcal{G}^c(G_2, r_j), \ldots, G^c_{r_j,\ell} = \mathcal{G}^c(G_\ell, r_j)\}, \quad \forall j = 1, 2, \ldots, k$. Simply put, for each graph in the training set and each coarsening ratio , a coarsened graph is generated. The node features in the coarsened graph are obtained by aggregating the features of the corresponding nodes in the original graph using methods like mean, max, or sum.

5.4.2 Regularization Loss

During training, the GNN is regularized by minimizing the CMD between the node embeddings of the original and coarsened graphs. The total loss function used for training is a combination of the standard supervised loss (e.g., cross-entropy for classification), denoted by $\mathcal{L}_{\text{orig}}$, and the CMD-based regularization loss:

$$\mathcal{L}_{\text{tot}} = \mathcal{L}_{\text{orig}} + \lambda \sum_{j=1}^{k} \sum_{i=1}^{\ell} \text{CMD}(f_\theta(G_i), f_\theta(G^c_{r_j,i})), \tag{5.2}$$

where λ controls the strength of the regularization, ℓ is the number of graphs in the training set, k is the number of coarsening ratios, and f_θ is the GNN with trainable parameters θ.

5.4.3 Pseudocode

In Algorithm 1, taken from [9], we can see the pseudocode of the regularization loss computation for a batch of graphs. In practice, the coarsened graphs can be pre-computed before the training begins and the computation is parallelized across graphs in the batch for efficiency. Authors report a 50% overhead in terms of training time with respect to the same model without SizeShiftReg.

Algorithm 1 Computing Regularization Loss for an Input Batch during Training

Require: Coarsening ratios C, coarsening function $\mathcal{G}^c(\text{graph, ratio})$
Input: Batch B of size n_b, GNN model f_θ

coarsened_batches $\leftarrow \{\}$
for r in C ▷ Create a new batch of coarsened graphs for each ratio
    Batch $B_r \leftarrow [\,]$
    for G in B
        $G^c_r \leftarrow \mathcal{G}^c(G, r)$
        B_r.add(G^c_r)
    end for
    coarsened_batches $\leftarrow$ coarsened_batches $\cup B_r$
end for
$\ell \leftarrow 0$ ▷ Initialize loss
for B_r in coarsened_batches
    for i in $\{1, 2, \dots, n_b\}$
        embs_og $\leftarrow f_\theta(B[i])$ ▷ Compute node embeddings for original graph
        embs_coarse $\leftarrow f_\theta(B_r[i])$ ▷ Compute node embeddings for coarsened version of graph
        $\ell \leftarrow \ell +$ CMD(embs_og, embs_coarse) ▷ Compute CMD between node embeddings
    end for
end for
return ℓ

5.5 Limitations

An important assumption for the effectiveness of SizeShiftReg is that small graphs contain information relevant for larger instances. However, this is not tied to this method specifically, it is also true for other approaches with similar goals [51, 52, 9]. In case there are no features that are independent of the graph size, we are left with including larger graphs in the training set as the best option.

6
Our Method

In this chapter, we motivate and describe our methodology that builds on top of the approaches in previous chapters (Chapter 4 and Chapter 5, in particular).

6.1 Motivation

In this section, we specify the problem setting that we focus on, followed by the gaps that we identified in the current approaches to solving it. Please note that the aim of this section is to motivate our approach by describing what type of impact we aim to make. That is, we do not get into the technical realization nor do we necessarily implement and test every idea mentioned in this section. The concrete description of what we have done to achieve our goals is left for the following sections of this chapter as well as Chapter 7.

As already mentioned in previous chapters, NAR architectures for the TSP have the following issues [4]:

- Training on large-scale instances is very computationally expensive.
- NAR architectures scale poorly to large instances.

We can try overcoming this issue by including larger instances in our training set and come up with methodological changes to enable more efficient training. However, such approaches in the literature typically require a lot of computational resources, complex models, and increased training time. Moreover, even if training on larger instances becomes the norm at some point, we still believe that generalization across sizes presents a fundamental property that models should have. For this reason, we focus on the zero-shot generalization setting.

One of the most successful (and simplest) approaches to achieve generalization of NAR models to significantly larger sizes is introduced in [8]. The authors combine the Att-GCRN architecture with compositional heatmaps to train on graphs of size 50 and generalize to sizes of 200, 500, 1000, and 10000 with a lot of success. In fact, most of the subsequent papers actually use the testing setting from [8] which shows its importance. For more details about these approaches, please refer to Chapter 3.

However, we believe that the proposed method has a few downsides that we try to improve. In short, the current approach trains a very complex model on a fixed, small graph size without any effort to make the model itself directly generalizable. Then, the model that overfits a specific size (and performs very well on it) is used to construct a compositional heatmap as described earlier. Thus, all the generalization effort is taken away from the model, that only focuses on a specific size, and transferred into the compositional heatmaps procedure.

It seems like the approach completely neglects the model's direct generalization ability and aims to solve the problem by going to the other extreme. In fact, the authors do not even report their model's performance without the compositional construction of heatmaps. From one point of view, this is reasonable because the performance is way better than previous methods trying to directly generalize the NAR approach. On the other hand, we think that designing simpler models with improved direct generalization (apart from being an interesting analysis on its own) can be combined with the compositional heatmaps approach. In the following subsections we discuss these considerations in more detail.

6.1.1 Overly Complex Model

The graph encoder in [8] is the Att-GCRN model described in subsection 4.2.2 with d (embedding dimensions of nodes and edges) set to 300 and ℓ (number of convolutional layers) equal to 30. The encoder is followed by a 3-layer MLP as described in subsection 4.3. This leads to a large model of just over $11{,}000{,}000$ parameters for relatively small graph sizes of 50. In this subsection, we discuss how a simpler model can provide benefits in terms of efficiency and the potential to extend the compositional heatmap construction.

Efficiency

A very obvious reason for trying to obtain a simpler model is the efficiency. It seems as, given that it only learns on the small graph sizes, the proposed Att-GCRN model might be too complex. Having a simpler model would allow for quicker inference, shorter training time, and reduced memory requirements. Of course, this could come with a certain degradation of performance. In the following sections regarding our solutions and experiments, we describe our attempts on finding a proper trade-off.

Increased Compositionality

Another benefit of having a simpler model that performs as well (or closely) to the more complex model is the opportunity to extend the compositional heatmaps procedure. With a smaller model, we could potentially have multiple levels of graph sampling instead of just one. As of now, the method works by splitting a large graph (e.g., 10000 nodes) into subgraphs of 50, uses the model to obtain heatmaps of each subgraph, and then merges them into a large heatmap. However, we could also split the same large graph into subgraphs of, e.g., 1000 nodes, then split them into subgraphs of 50, and repeat the heatmap merging procedure twice. Of course, this could theoretically be done with any model, but the time and memory constrains would force us to use a simpler model. That is, we use the example of the increased compositionality to illustrate that a lightweight model could give us the opportunity to extend our techniques within the same resource setting.

6.1.2 Model Designed for Fixed Graph Sizes

As we already mentioned, the authors of [8] state that they do not intend to make a model directly generalizable across sizes. However, it could be interesting for the following reasons:

- Analyzing the influence of regularization techniques applied to NAR architectures in the combinatorial optimization domain
- Combining the regularized model with the heatmap compositionality approach.

Regarding the first point, we are interested in how SizeShiftReg [9], originally designed for the graph classification task, extends to learning the TSP. Secondly, if we had a model that produces high-quality heatmaps for different sizes, we could sample the graphs of various size at inference time instead of fixing on the training size. In this way, we hope to better identify the promising edges across various graph sizes. Moreover, we could potentially increase the graph sampling size, up to the size on which the model performs well, while still training on small graph sizes.

6.2 Technical Changes

Now that we stated what we aim to achieve, we are ready to present our technical contributions. Firstly, we present our decision regarding the model architecture. Then, we explain why we opted for the SizeShiftReg and how we adapted it to our task.

6.2.1 Model Architecture

We combine the approaches from [4] and [8]. That is, we took the AGCN from [4] (described in 4.2.1) and adapted the compositional construction of heatmaps to work with this architecture. In this subsection, we discuss the reasoning behind the architecture choice followed by the implementation challenges.

Why do we change the architecture?

We decided to use the AGCN architecture from [4] because it is suitable for our approach both from the methodology and analysis point of view. AGCN performs anisotropic message passing using edge gates $\sigma(\boldsymbol{e}_{ij}^{(\ell)})$ to scale the neighboring node contributions. Thus, the only role of edge representations is to weigh the neighborhood in the aggregation function. Notice how the final MLP layer is not defined on edge representations, but on the corresponding node embeddings. On the other hand, the Att-GCRN [8] performs the prediction directly on edge representations (and it also uses them to learn the attention coefficients $\alpha_{ij}^{(\ell)}$). If we put this into the context of applying SizeShiftReg, we think that the architecture where node embeddings play a key role for final predictions could be more suitable. Moreover, regarding our goal to make the architecture more generalizable, notice that AGCN introduces the global information into the edge probability prediction while the Att-GCRN focuses only on the particular edge.

Additionally, the AGCN architecture also provides a stronger foundation for our analysis. The papers [4] and [8], even if dealing with the same problem, essentially have different goals in mind. While [4] tries to systematically analyze generalization capabilities of GNNs for learning the TSP, [8] deals directly with trying to obtain the best performance possible. For this reason, [4] provides a lot of experimental insight into the architecture behavior (even if mostly concerning the AR approach) under various settings while [8] talks about a single setting that achieved state-of-the-art performance at the time. Thus, we can take advantage of the insight provided by the authors of [4] to extend the analysis of NAR architectures. In fact, our application of SizeShiftReg was motivated partially by the results of their analysis as explained in subsection 6.2.2. Finally, [4] is an extension of the work in [23] (done by mostly the same authors of [4]) where Att-GCRN (with subtle differences) was initially used for the same task. Thus, the authors themselves have chosen the AGCN architecture for the generalization performance analysis.

Implementation Details

From the conceptual point of view, applying the compositional heatmap construction to a new architecture was simple while the code implementation posed a few challenges.

In principle, the output of both architectures is a heatmap representing the probability of each edge belonging to the optimal TSP solution. However, there is a difference regarding the model inputs. Given that we use graph sparsification as opposed to Att-GCRN that operates on full graphs, we had to consider two different scenarios at inference time. Before explaining that, it is important to understand that the architecture we use predicts the probabilities even for the edges that were removed in graph sparsification. That is, absence of an edge will affect node embeddings by not considering the neighbor connected by it, but it is still possible to assign high-probability to that edge if the model learns to do so. Thus, we have an option of setting the probabilities of such edges to zero before merging the heatmaps. This decision introduces a trade-off between exploration and exploitation, and we decide to experiment with both alternatives. In the experiments, we will refer to it as *heatmap masking*.

For the implementation, we combined the code repositories of the corresponding papers. Both of the models are implemented using PyTorch [53]. Moreover, we want to emphasize that the repositories did not originally work with the latest version of PyTorch and other supporting libraries. We have made simple changes to upgrade the versions which led to a significant training time reduction with respect to the one reported in [4]. In particular, the training time on the same model is approximately 2.5 times faster. To be clear, we have just updated the code to work with the newest versions of libraries without making any significant changes to optimize the training. In this thesis, we report the reduced training times.

6.2.2 Adapting SizeShiftReg

Firstly, we explain why we have chosen SizeShiftReg [9] as our regularization technique. Then, we explain how we applied it to our task.

Motivation

SizeShiftReg has been successful in the generalization setting of the graph classification task. In particular, the authors [9] trained on the 50% smallest graphs and tested on 10% largest graphs in the dataset. They obtain up to 30% average performance increase on graphs 3 to 9 times larger than the average training size.

If we put this into our context, where we train on size up to 50 and test on sizes up to 10000, it is obvious that the size difference can be way larger. However, we find SizeShiftReg suitable for two main reasons that we already briefly mentioned:

- NAR models in the literature have usually not been successful in generalizing directly (without divide-and-conquer techniques such as compositional heatmaps and/or complex post-hoc search methods) even to graphs 3 to 9 times larger. We want to test the direct generalization capabilities of NAR approaches as well as the applicability of SizeShiftReg to different domains. Moreover, SizeShiftReg could also be applied to AR techniques that usually aim for direct generalization. From this point of view, probing into its performance in other domains is an interesting research question.

- Strong generalization to larger sizes, even if "only" 3 to 9 on average, can be combined with the compositional heatmaps method as we described before. That is, at inference time, we could sample larger sizes than the ones we trained on and/or sample various graph sizes to obtain relevant edges across different scales.

We argue that the improvements that SizeShiftReg could bring would be valuable in our setting, but why would we expect it to perform well for learning the TSP? First of all, if we take a more detailed look at how a GNN is used in this approach, we understand that there are many similarities to the classification task. That is, we do not directly optimize for the shortest cycle, but we predict the probability of each edge. Thus, we can see this as estimating the probability of each edge being classified as an edge that belongs to the optimal tour. More importantly, conclusions from analysis in [4] align with the purpose of SizeShiftReg. Before we describe it, we note that it was conducted for AR approaches, but we believe that similar behavior should be observed for NAR approaches as well.

The authors of [4] analyze how distributions of node and graph embeddings behave across different graph sizes. They compare the embeddings of the best and worst performing architecture settings (aggregation functions and normalization techniques) in terms of generalization performance. Their results show that the best-performing architectures lead to the most stable embedding distributions across different graph sizes. Maintaining similar representational distributions across different sizes shows that the network is extracting meaningful information from training instances [4]. Thus, the authors advocate for development of GNN architectures and normalization layers that are expressive, but invariant to distribution shifts at the same time. We identified SizeShiftReg as a promising technique for enabling the network to learn these meaningful representations across scales as well as to prevent performance breakdowns due to distribution shifts.

Implementation

We said earlier that we use graph sparsification, not only for efficiency and generalization purposes, but also for a proper application of SizeShiftReg. We need to redefine input graphs for a meaningful application of graph coarsening. That is, we aim to obtain a smaller representation of an original graph preserving its main topological features. For this reason, coarsening a fully-connected graph would not be a very meaningful choice. Instead, graph sparsification introduces interesting topological properties by reducing the number of edges.

We perform graph sparsification by, instead of a full graph, connecting each node only to its $n \times k$%-nearest neighbors (k is a parameter) as explained in Section 4.1.1. Notice that such graph sparsification does not necessarily lead to an undirected graph. This introduces another design choice for the application of SizeShiftReg. It is originally implemented only with undirected graph coarsening techniques, i.e., it handles directed graphs by first making all the edges undirected and then performing graph coarsening. This is important as most of the graph coarsening techniques are actually designed for undirected graphs and they require symmetrical adjacency matrices. If we were to just blindly apply this, we could obtain (especially for large coarsening ratios) a coarsened version of the graph with more edges than the original. We do not want this behavior, thus we are left with the decision of applying directional graph coarsening techniques or heuristically fixing the coarsened graph. As the authors demonstrated the robustness of SizeShiftReg to different graph coarsening techniques [9], we opt for the latter. Designing a specific directional graph coarsening technique could be an interesting direction for future work, but we do not expect it to have a significant impact on our results.

When we consider alternatives for reducing the edge count of coarsened graphs, we should choose the one that preserves the topology and edge distribution in the best way. Hence, we keep the topology obtained from the coarsening procedure and we operate on each node independently by removing its edges, so that it keeps $n \times k\%$ of edges. Basically, if the conversion to an undirected graph has led to the coarsened graph having some nodes with higher percentage of edges than in the original graph, we remove the furthest edges until the percentages are equal. In this way, we take into account the topology induced by graph coarsening, but also align the edge count distribution across original and size-shifted graph versions. Lastly, we note that we always use the mean function to aggregate the coarsened graph node features.

Another important design decision is specifying the embeddings that we regularize. In [9], the authors use a basic GCN version (along with some other GNN architectures) where they only have the node embeddings while our anisotropic architecture produces edge embeddings as well. However, we still decide to calculate the regularization loss based only on the CMD between node embeddings of original and coarsened graphs. As we explained earlier, the edge embeddings are only used for weighing the neighboring node contributions, thus it is not clear whether regularizing them would be a valid approach. Moreover, there is a possibility of regularizing the graph embeddings, but, as we use very simple graph pooling techniques (e.g. sum, mean, max), the graph representations directly depend on node representations. Introducing a learnable graph pooling technique along with SizeShiftReg would be an interesting direction to enhance the influence of global representation, but we restrict to node embeddings in our work.

However, the biggest challenge in our SizeShiftReg application was the increase in dataset size with respect to the original paper. In particular, authors of [9] train on datasets containing up to $9{,}629$ graphs while both of the papers we build upon use way larger datasets. In particular, the TSP20-50 dataset from [4] contains 1,280,000 instances while [8] uses 990,000 instances of the TSP50 dataset. Not only are the datasets themselves large, but they can also get multiple times larger depending on the number of coarsening ratios used. This large difference in scale introduces the following challenges:

- Increased time for dataset preparation as we have to coarsen a large number of graphs
- Increased model training times
- Increased GPU and CPU memory requirements.

For these reasons, we have not used the full datasets as in [4, 8] for our SizeShiftReg experiments. We report the details about the dataset sizes we used in Section 7.3.3.

Finally, we want to briefly comment on the practical code implementation. We had to combine code repositories from [4] and [9] that use different libraries for data loading, data preparation, and model training. Moreover, given the aforementioned problem of large datasets, we had to increase the computation time to save memory in certain parts of the implementation. Details are reported in the section discussing the SizeShiftReg training time overhead.

7 Experiments

7.1 Experimental Setup Overview

Our experiments are conceptually split based on the following considerations:

- Architecture experiments vs SizeShiftReg experiments
- Direct generalization vs Compositional generalization.

To make things clear, all of these aspects are interconnected, but we explain them separately to provide the reasoning behind the experimental design. We assume the knowledge of the previous chapter because we aligned the experiments with the goals of our analysis.

In the architecture experiments, we vary the model size, activation functions, normalization techniques, depth, and width (along with additional hyperparameters). These aspects are important to find a simpler architecture that performs as well (or close to) the complex Att-GCRN architecture [8]. Moreover, different activation functions and normalization techniques influence the distribution and magnitude of node embeddings which is especially relevant in the context of applying SizeShiftReg. These aspects are important for the generalization of any model, but aligning node embeddings using CMD requires them to be represented as bounded probability distributions. To sum up, the goal of these experiments is to analyze the architectural changes independently, but also as a preparatory analysis for SizeShiftReg application.

Regarding the second categorization, we analyze two different task settings. For direct generalization, we use the model as in [4]. That is, the model is used directly on larger instances and the heatmap is explored by beam search. Given that the beam search is a very simple technique, we somewhat mimic the generalization setting from [9], even though we do not really perform direct classification as we have a post-hoc search. On the other hand, compositional generalization is tested following [8]. We sample small graphs, use the model on them, combine the heatmaps, and run MCTS to obtain the final solution. In this way, we test how our simpler models compare to the Att-GCRN architecture.

Putting it all together, we provide the structure of our experiments. The authors in [8] have only used their model with the compositional approach without providing any insight into the model's direct generalization performance. Instead, we use the direct generalization results to guide our model choice for the compositional evaluation. In order to understand how we can combine the two approaches, we need to explore the connection between them. Thus, we organize our direct generalization experiments in the following way:

- Show the main challenge of direct NAR generalization for TSP
- Analyze the influence of simple architectural changes
- Analyze the influence of SizeShiftReg.

Finally, we use these models in the compositional generalization setting.

7.2 Experimental Setup Details

7.2.1 Training

Datasets

We train on TSP datasets provided in [4], containing synthetically generated graphs with n nodes uniformly sampled in the unit square. Models without SizeShiftReg are trained on 1,280,000 TSP instances of varying sizes from 20 to 50 (named TSP20-50). On the other hand, models with SizeShiftReg are trained both on graphs of varying sizes (20-50) as well as graphs with a fixed size of 50 to test its influence in different settings. Moreover, for SizeShiftReg models, we do not use the whole dataset, but up to 384,000 instances because of time and memory constraints. This will be clearly explained in the dedicated sections.

Model Setup

As already described, we will train the AGCN architecture. In the original paper [4], they focus on the model with 4 GNN layers and hidden dimension $d = 128$ (although they do test other settings as well) which amounts to approximately $350{,}000$ parameters. We vary the number of GNN layers between $4, 7, 10, 15, 18, 27$, and 30 and embeddings dimensions between $64, 96$, and 128. Models are trained for 10 or 20 epochs using $Adam$ optimizer with a fixed learning rate of $1e-4$, and a batch size of 128.

We always use max aggregation for node embeddings, mean aggregation for graph pooling, and 0.5% as the percentage in graph sparsification. On the other hand, we test various normalization techniques, i.e., batch normalization, layer normalization, L_1 normalization, and L_2 normalization. Regarding the activation functions, we test ReLU, tanh, and sigmoid.

Unless we state differently, we consider the model configuration that was adopted in [4]. That is, we set the hidden dimension d to 128, we use ReLU as the activation function, and batch normalization with batch statistics. We will explicitly state the other hyperparameters, but we specify default options to avoid redundancy.

7.2.2 Evaluation

Our methods are evaluated based on the *optimality gap*. Optimality gap is defined as the average percentage ratio of predicted tour lengths relative to optimal tour lengths. We calculate it based on the formula from [23] that is used through the literature:

$$\frac{1}{g}\sum_{i=1}^{g}\left(\frac{L(\pi_i^{\text{pred}})}{L(\pi_i^{\text{opt}})}-1\right), \tag{7.1}$$

where g is the test dataset size while $L(\pi_i^{\text{pred}})$ and $L(\pi_i^{\text{opt}})$ denote the length of the predicted and the optimal tour for the i-th test instance, respectively.

For all the sizes, but the TSP10000, we use the Concorde solver [54] to obtain an optimal solution. The TSP10000 solutions are compared to the LKH heuristic [55] because it would take too much time to compute the optimal solution.

Beam Search

For the direct beam search evaluation, we follow the setting from [4]. That is, we test our models on a test set of 25,600 instances (taken from [4]), consisting of 1,280 samples for each size from 10 to 200. We fix the beam search size to 128 and report the optimality gap across sizes. The results will be reported on the plots with TSP sizes on the x axis and optimality gaps on the y axis as in [4]. However, we keep the constant scale on the y axis ranging from 0% to 100% for all the plots instead of cutting off at the largest value for each plot separately. We do this to ensure that our results are comparable across different plots which is very important to properly understand our analysis.

MCTS

For MCTS evaluation, we follow the setting in [8], reusing all their test datasets. That is, we test on 128 instances of TSP200, TSP500, and TSP1000 while we use 16 samples for TSP10000. For a fair comparison, we set all the hyperparameters as in the original paper. Most importantly, we set the sampled graph size to 50, and we fix the amount of search time as in the paper. However, there is an important note regarding the runtime. In [8], they use the GPU implementation of the MCTS and report the corresponding search times. We ran into many problems trying to reproduce these results on the GPU. All the other works in the literature that we are aware of [31, 6, 7, 29] have opted for the CPU version, probably for the same reasons. CPU runtimes are usually reported on the whole test set by parallelizing on many cores, but we decided to follow [29] and report the running time per instance. In this way, we have a fair comparison that does not depend on the number of CPU cores used. We also report the heatmap generation time as in all the other works.

7.2.3 Hardware resources

For model training, we used GPUs available in free versions of Google Colab and Kaggle Notebook. Moreover, we had access to 2 NVIDIA Titan Xp GPUs with 12 gigabytes of memory each. For the MCTS experiments, we used one Intel Xeon Gold 5220 CPU utilizing 64 cores.

7.3 Direct Generalization Results

In this section, we report beam search experiments conducted to probe deeper into direct generalization capabilities of NAR approaches. We already mentioned that the authors of [4] have done a thorough analysis using the autoregressive AGCN architecture. However, they only briefly dealt with NAR approaches as they argued that autoregressive decoding is a better inductive bias for the TSP. On the other hand, we already discussed in our motivation section how NAR approaches are still worth exploring because they are much faster and can be used to compositionally construct the heatmaps. For these reasons, we decided to extend the existing analysis of direct NAR generalization.

7.3.1 What is the main challenge?

Firstly, we want to show why the current NAR approaches have not been successful in direct zero-shot generalization. As we will see, they tend to overfit on the training data size which completely undermines their ability to learn the underlying generalizable patterns. Of course, this is common for any zero-shot generalization setting, but we provide the results to illustrate how drastic this can be for NAR approaches applied to TSP.

We train three models, test them using beam search, and plot the optimality gaps as a function of TSP size. All three models have the AGCN architecture, but they have varying complexity. The first two models are trained on all 1,280,000 instances of sizes from 20 to 50 by setting the number of GNN encoder layers to 4 and 10. We name them AGCN4 and AGCN10, respectively. The third model, named AGCN*, also has 4 encoder layers, but it is trained only on a subsample of 128,000 TSP20-50 instances. AGCN4 is trained for 10 epochs, AGCN10 for 20, and AGCN* for one epoch only. The models with 4 layers have around 350,000 parameters, while the 10-layer model has around 850,000.

The results are plotted in Figure 7.1. We first focus on the difference between AGCN4 and AGCN10. We can see that the more expressive model fits the training data way better, but its generalization performance deteriorates as the testing size increases. It is true that it performs better even outside the training size, but the relative improvement decreases until the simpler model eventually gets better. This behavior is common amongst NAR architectures and presents an issue that is very hard to overcome. In fact, we added the AGCN* specifically to emphasize how drastic this can be. As you can see, the model that is trained only for one epoch on only a tenth of data significantly outperforms the other two models on test instances with approximately 130 nodes and above (around 2.6 times increase with respect to training graph sizes).

Basically, we performed slightly modified experiments to restate what the authors in [4] have already concluded. That is, the current NAR approaches cannot directly adjust to the distribution shift of larger test instances. At that point, they mainly focused on the alternative autoregressive approach. Instead, we extend the analysis of NAR approaches in the remainder of this section.

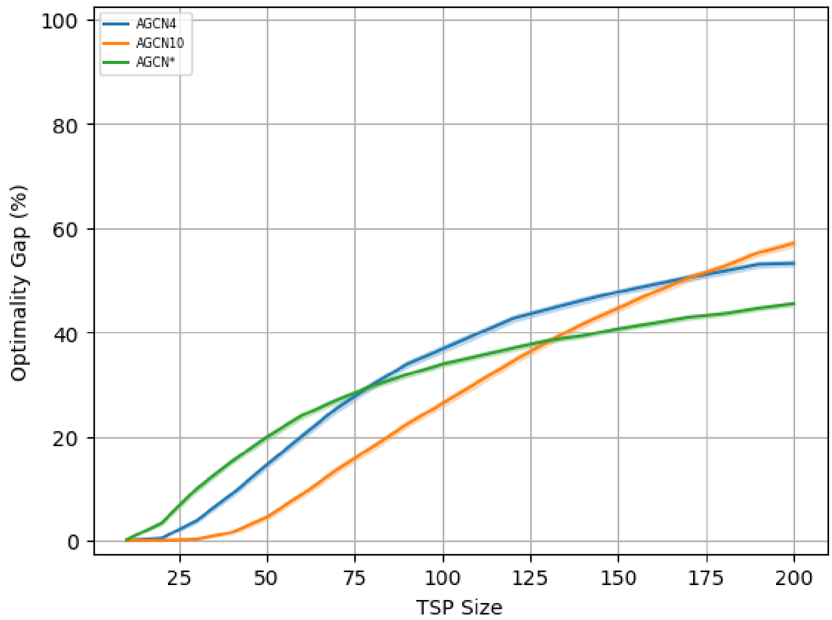


Figure 7.1: The main challenge of direct NAR generalization. Increasing model complexity and number of training instances leads to overfitting on the training graph sizes.

7.3.2 Effects of Simple Architectural Changes

The current analysis of NAR approaches does not cover the influence of a few components that are crucial for our method. We are interested in the effects of different normalization techniques that bound the node embedding values. Moreover, we want to analyze the influence of network's depth and width on the direct generalization performance.

Please notice that these considerations are important on their own, but they also provide insight into the SizeShiftReg application. That is, they influence the node embedding distribution which is crucial for the CMD-based regularization loss. In fact, we cannot directly apply SizeShiftReg in the original setting because it does not produce the node embeddings in the form of a bounded probability distribution. The embeddings' scale is controlled by the learned parameters in the batch norm while the ReLU activation function also has an unbounded output. In that case, the cross entropy loss and the regularization loss have very different scales which makes it very hard to analyze and tune the regularization influence. We will get back to these points when we report SizeShiftReg results.

In Figure 7.2, we can see the effects of different normalization techniques on direct generalization performance. We train five AGCN models on all $1,280,000$ instances of sizes from 20 to 50. We set the number of GNN encoder layers to 10, and number of epochs to 20. The baseline is the model with the unchanged architecture (named "original"), and all the other models introduce simple changes to assess their influence. That is, "tanh" uses the tanh activation function instead of ReLU for node embeddings (edge embeddings still have ReLU). The model named "L_1" normalizes the node embeddings to have L_1 norm equal to one while the "L_2_batchnorm" uses the L_2 norm. Finally, "L_2_no_batchnorm" uses L_2 norm, but also removes the batch normalization.

The most important thing to keep in mind when analyzing these models is the trade-off between the training size performance and the generalization performance. That is, improving one of the two is fairly easy, but improving both poses quite a challenge. The "L_1" has no desirable properties, thus we totally excluded it in further analysis. "L_2_no_batchnorm" generalizes the best out of all the models, but it fails to capture the training set patterns. It could be interesting scaling it up further because it does provide the best overall trade-off amongst sizes. The L_2 model with batch norm does not generalize as well as the original, but it stabilizes the learning throughout the epochs (not shown in the plots) and allows the model to better capture the training data size patterns. Finally, the "tanh" model outperforms the baseline on all the sizes, so we explore it further.

Figure 7.3 shows the optimality gap on the validation set with fixed graph size 50 during model training. We have the training steps on the x axis and the optimality gap value on the y. The blue line represents the model with tanh on the node embeddings while the green line shows the original model. We can see that the tanh function enables the model to capture the training size patterns in a better way while maintaining stronger generalization capability.

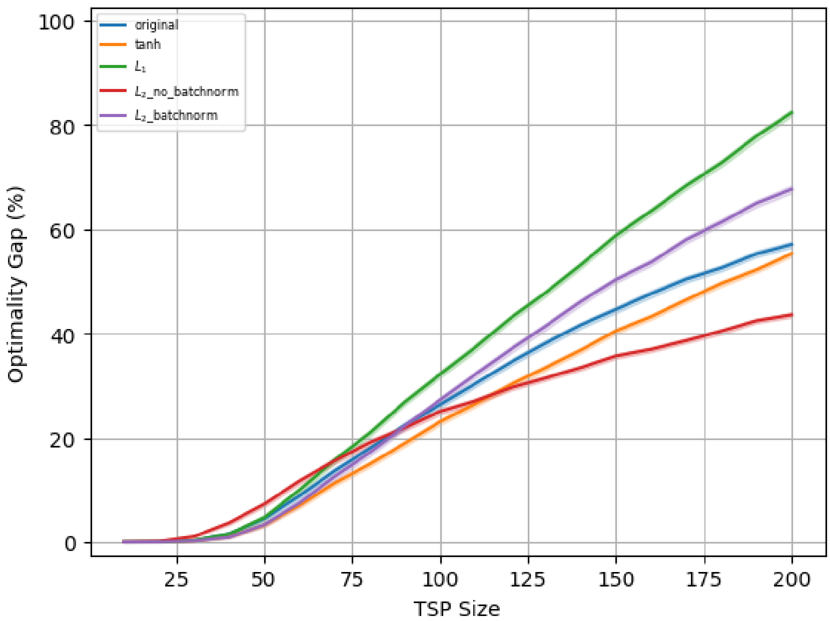


Figure 7.2: Effects of various normalization techniques on direct NAR generalization.

That being said, the tanh model still has poor generalization overall. Bounding the node embeddings within a certain range can definitely have a positive impact on generalization as we can see in the two of our models. It is possible that an extensive analysis of classical normalization techniques could have significant impact on the generalization. However, we instead use this preliminary analysis to guide further decisions regarding the compositional approach and the application of SizeShiftReg.

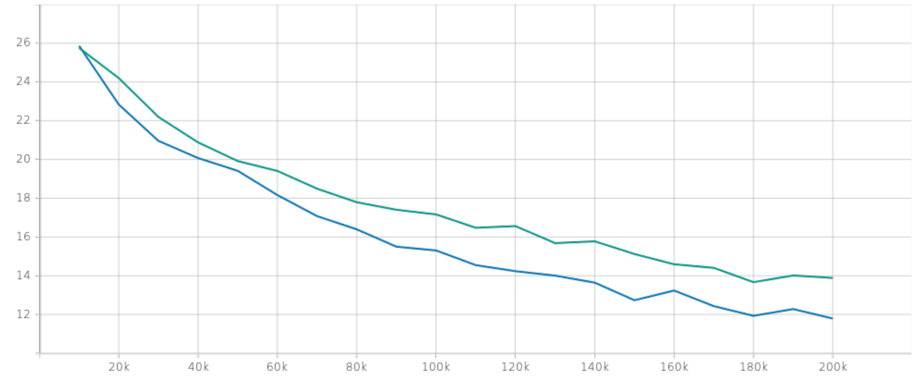
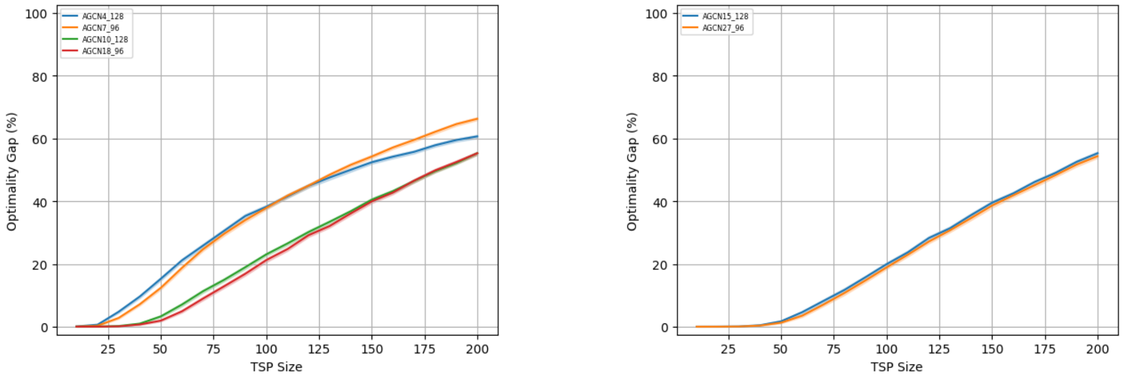


Figure 7.3: Validation optimality gap of AGCN models on graphs of size 50. The green line represents the optimality gap of the model using ReLU (original), while the blue line refers to the model with tanh activation function. Bounded node embeddings obtained by using the tanh activation function enable the model to capture the training size patterns in a better way while still maintaining stronger generalization capability.

We also test the influence of varying the model depth and width. Depth refers to the number of GNN encoder layers ℓ while width refers to the embedding size of nodes and edges, denoted by d. We train models on three levels of complexity, i.e., approximately $350{,}000$, $850{,}000$, and $1{,}280{,}000$ trainable parameters. For each level of complexity, we train two different versions of the model by changing the depth and width, keeping the same number of parameters. All the models have the AGCN architecture with tanh activation function because it had the best overall performance in the previously reported analysis. We name them by putting the number of layers after the architecture name (as before) followed by the embedding size after the underscore. For example, AGCN4_128 has 4 GNN encoder layers and the embedding size set to 128. Results are shown if Figure 7.4. Please note that we separated the results on two plots only for better visibility. That is, putting all the lines in one plot blurs the small differences that we want to interpret.

Overall, we notice that the performance remains similar with different depth/width configuration. Simply changing the model depth does not solve the generalization issue. However, as noted earlier, improving the performance both on the training size and outside of it is not straightforward. Thus, we want to take into account even the slight improvements. In particular, deeper models perform better on the training sizes on all three complexity levels. Moreover, the depth improves the generalization performance on medium and large models while the shallower model has better generalization on the lowest scale.



(a) Small and medium scale models

(b) Large scale model

Figure 7.4: Effects of varying model depth and width across different model complexity levels.

To sum up, simple architectural changes can mitigate the generalization problem to a certain degree. However, it seems that more significant improvements require the application of advanced regularization techniques. We use the insight obtained in this analysis to guide the application of SizeShiftReg [9].

7.3.3 Effects of SizeShiftReg

In this section, we present the results of applying SizeShiftReg [9] to our models. We begin by summarizing key insights that are crucial for interpreting the results. Next, we present a simple example as a preliminary analysis. Following this, we test the effects of SizeShiftReg when training both on variable and fixed graph sizes. Finally, we report the training time overhead of using SizeShiftReg.

Summary

As we explain at the end of Section 6.2.2, using all 1,280,000 training instances in SizeShiftReg experiments requires more resources than we have available. For this reason, we use up to 384,000 instances of both TSP20-50 and TSP50. This amounts to 30% and 39% of dataset instances used in [4] and [8], respectively. Furthermore, instead of testing various graph coarsening techniques as in [9], we always use the spectral graph coarsening [5]. Even if we do not use the whole datasets, we believe that these dataset fragments are large enough to perform a meaningful analysis.

Taking this into consideration, the models reported in this section are not able to reach the competitive performance on training graph sizes with respect to some of the models using all the training data. For this reason, they have not been considered in the final compositional generalization analysis where we try to obtain results comparable to [8]. However, we have used enough data to show the potential of SizeShiftReg in our application. In particular, we obtain up to 7.4% reduction in the average optimality gap on the test dataset while improving (or at least maintaining) the performance on every graph size. We have to introduce an additional consideration to explain the significance of this result. That is, a decrease in the average optimality gap on all the test sizes can be obtained by improving the performance on some sizes and degrading it on others. Such results are valuable as long as the relative performance decreases are not large. For this reason, we do not simply provide the numerical results, but we interpret the plots showing the model performance. However, more significant results are obtained when we are able to decrease the average optimality gap while not increasing it for any size. We were able to obtain this in our best setting as already mentioned. However, scaling such results to more complex models and more data requires further work.

Finally, we want to add the note regarding the influence of bounded node embeddings. As we report in Section 7.3.2, restricting node embeddings in different ways can improve generalization performance. In our SizeShiftReg experiments, we always bound the node embeddings because this is necessary for its application (explained in Section 5.3). The baseline models are identical in every aspect, with the only difference being whether SizeShiftReg is applied during training. This means that the generalization performance improvements that we report are not the result of adding normalization and/or changing the activation functions, but due to the application of SizeShiftReg.

To sum up, using SizeShiftReg to improve the performance of current state-of-the-art models in the literature requires more time, resources, and experimentation than available for this thesis. However, we believe that our SizeShiftReg results provide a meaningful proof of concept for its application in learning the TSP. Our results are promising, but we cannot be sure whether they would translate to a larger scale without additional exploration. Moreover, as you will see in Section 7.4, we were able to obtain strong results even without incorporating SizeShiftReg in the best-performing models, that is left for future work.

Preliminary analysis

We start by analyzing the effects of SizeShiftReg on a very simple model to understand its effects in a scenario where there is a lot of potential for improvement. That is, we train AGCN model with 4 encoder layers on 128,000 TSP20 instances for 10 epochs. We compare its performance with and without SizeShiftReg. In particular, we train one model with coarse ratio 0.8, one with 0.9, and one with both of them. We differentiate the models with and without SizeShiftReg by including "with_ss" and "wo_ss" in their names on the plot legend. For the model without SizeShiftReg, we report the results after training for 1 epoch ("epoch0" on the plot) and for 10 epochs ("epoch9"). For the SizeShift models, we always report the results after the 10-th epoch. Results are reported in Figure 7.5.

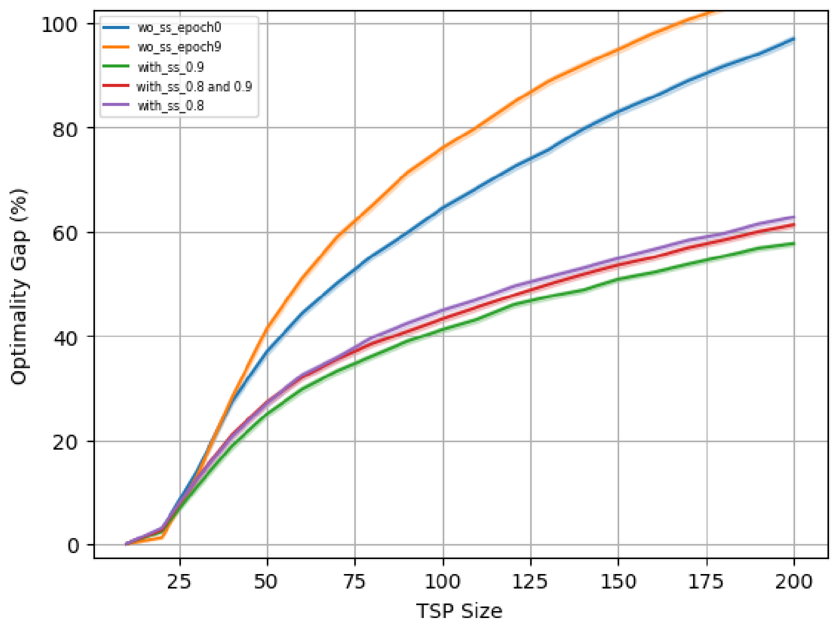


Figure 7.5: Results of preliminary analysis. SizeShiftReg brings major improvements on the simple model trained on small graph sizes of 20 nodes.

We can see that SizeShiftReg brings major improvements in terms of generalization performance in this simple scenario. In particular, the baseline model has very unstable training shown by the better performance after 1 epoch than after 10 epochs. SizeShiftReg introduces stability and allows the model to learn throughout the epochs. However, we want to point out that adding the coarsening ratio of 0.8 in addition to 0.9 brings down the performance. We will discuss this in more detail when we analyze more complex models in the following sections.

We designed this toy example to indicate the potential of SizeShiftReg in our application. However, obtaining significant improvements in more-realistic scenarios with better performing models requires carefully choosing various hyperparameters. Instead of extensively trying all the possible combinations, we tune them by tracking the model training behavior and/or considering previous results. We report this analysis in the following sections.

Training on varying graph sizes

We train AGCN models with 10 encoder layers and the hidden embedding size of 128 (resulting in approximately 850,000 parameters) on 384,000 TSP20-50 instances for 20 epochs. Since the models use ReLU activation function, we normalize the node embeddings to have a unit L_2 norm after each layer. We train the model without SizeShiftReg (named "baseline"), three SizeShiftReg models with coarsening ratio 0.9 varying the regularization coefficient λ between 1.0, 5.0, 25.0, and one model with both 0.8 and 0.9 coarse ratios with λ set to 5.0. Models are named based on the coarsening ratios and regularizing coefficient they use. We show the results in Figure 7.6.

We can see that varying the value of λ significantly changes the influence of SizeShiftReg. The model with λ set to 1.0 performs similarly to the baseline with small generalization improvements while increasing it to 5.0 leads to significantly better performance on larger sizes. However, it also decreases the performance on the training data size. Increasing λ to 25.0 leads to the model over-focusing on the regularization loss and increases the optimality gap on all the testing sizes. Finally, we see that adding the coarsening ratio of 0.8 in addition to 0.9 has a negative impact on the model performance.

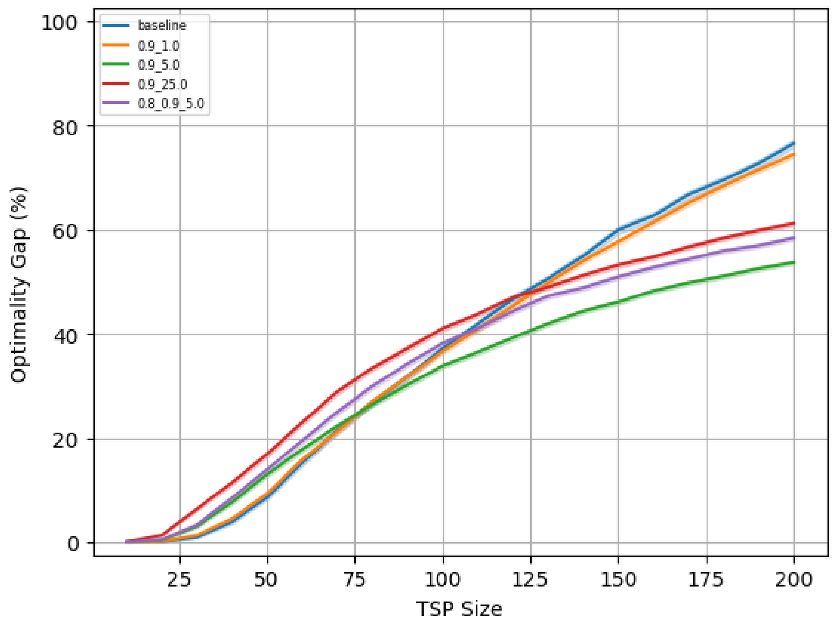


Figure 7.6: SizeShiftReg applied during training on the dataset with graph sizes from 20 to 50. Varying the normalization coefficient λ has a strong influence on the model performance. Training with λ set to 5.0 significantly improves model's ability to generalize, but negatively impacts the performance on the training data sizes.

We trained many more models in the same scenario (in terms of training data and model complexity) varying the normalization techniques and activation functions to test the effects of different node embeddings distributions on the improvements obtained by SizeShiftReg. The results exhibit very similar patterns as in Figure 7.6. That is, we are able to obtain significant improvements on large sizes by training with SizeShiftReg using the coarsening ratio 0.9 and tuning the normalizing coefficient. However, we always see worse performance on the training size and negative impact of lower coarsening ratios, i.e., 0.7 and/or 0.8. For this reason, we find that reporting all such results is redundant. Instead, we first analyze this behavior in more detail followed by a slightly changed setting where we obtain better relative improvement.

We train the same model as before with SizeShiftReg using coarsening ratios of 0.7, 0.8, and 0.9 together while λ is set to 1.0. It gets compared to the model without SizeShiftReg and to the model using only the coarsening ratio of 0.9 with the same λ. Figure 7.7 shows the performance while Figure 7.8 shows the training losses of the model using three coarsening ratios.

In Figure 7.8, we can see the cross entropy loss and the CMD loss (SizeShiftReg loss) throughout the training steps. The smoothed curves are represented in a strong color, whereas the originals curves have a faint appearance. The plots indicate that the model training achieved what we aimed for, i.e., it reduced both the losses in a reasonably stable manner. This means that we were able to introduce the SizeShiftReg effect into the model without hurting the performance on the original task significantly. However, in Figure 7.7 we can see that the model with three coarsening ratios performs worse than the baseline on all the sizes. Moreover, we also add the model with coarsening ratio 0.9 and the same value of λ as a reference. It seems that the larger coarsening ratios do not introduce a proper bias into the model.

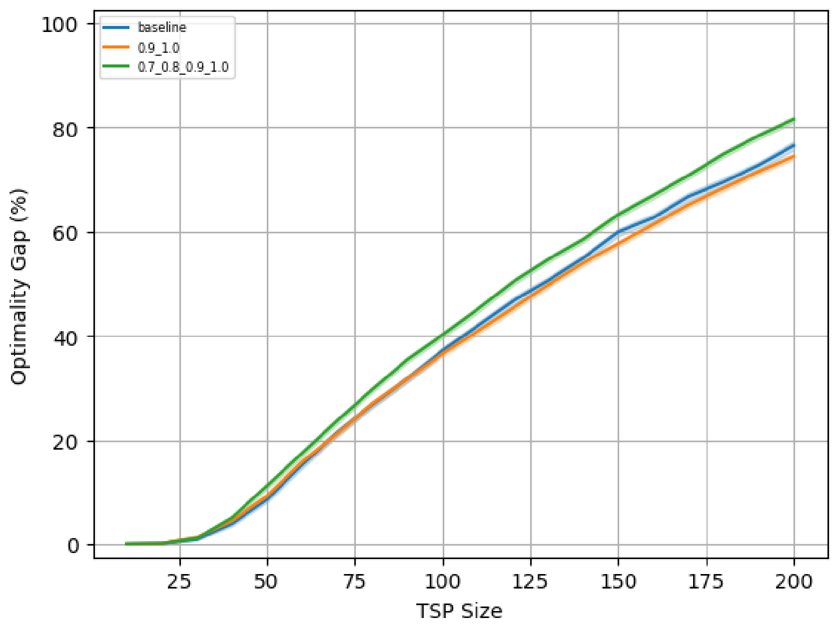
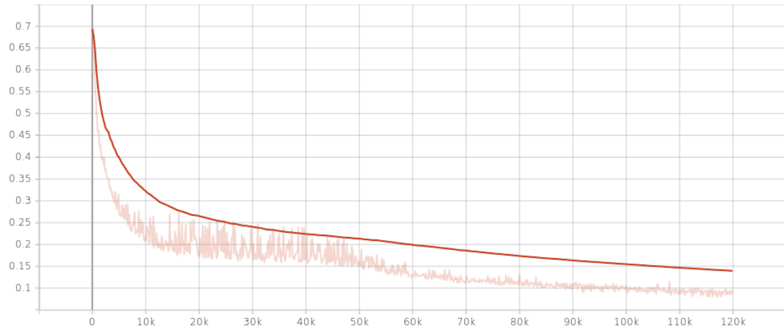
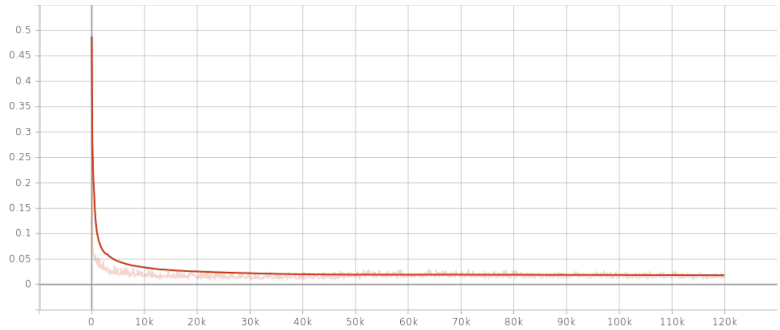


Figure 7.7: Lower coarsening ratios than 0.9 negatively influence the performance on all sizes even if the training is stable. Identifying the cause of this issue requires deeper analysis. It could be due to model hyperparameters, small training graph sizes and/or coupling between graph sparsification and graph coarsening techniques.



(a) Cross-entropy loss



(b) CMD loss

Figure 7.8: Cross-entropy and CMD loss during training. Original curves have a faint appearance while the smoothed versions have a stronger color. Both of the losses decrease in a stable manner which indicates that the hyperparameters are properly set for the application of SizeShiftReg.

One of the possible reasons for this might be that the lower coarsening ratios could lead to uninformative graphs. For example, coarsening the graphs of size 20 with the coarsening ratio of 0.7 produces graphs with only 14 nodes. Authors of [9] have noticed similar behavior in some of their experiments. However, this is just a conjecture, and understanding this behavior would require a deeper analysis. It could also be due to the model complexity that is not able to capture the patterns across various sizes with a limited number of parameters ($850{,}000$). Moreover, we could test the influence of removing more (less) edges in the graph sparsification technique. This would induce a different topology that could be crucial as the input to graph coarsening.

To show that even better improvements can be obtained, we provide a slightly different setting where SizeShiftReg shows strong generalization improvements without increasing the optimality gap on the training set sizes. We keep everything the same as before except for the model complexity and depth/width ratio. That is, the number of encoder layers is set to 30 and the embedding size to 64, leading to 640,000 parameters. The baseline model is compared to two models with SizeShiftReg. Both of them have λ set to 5.0 while the coarsening ratios differ between only 0.9 and $(0.8, 0.9)$ combined. Results are reported in Figure 7.9. The results show that the generalization performance is improved keeping approximately the same optimality gaps on the training sizes. Moreover, the combination of $(0.8, 0.9)$ coarsening ratios slightly improves on top of the 0.9 alone.

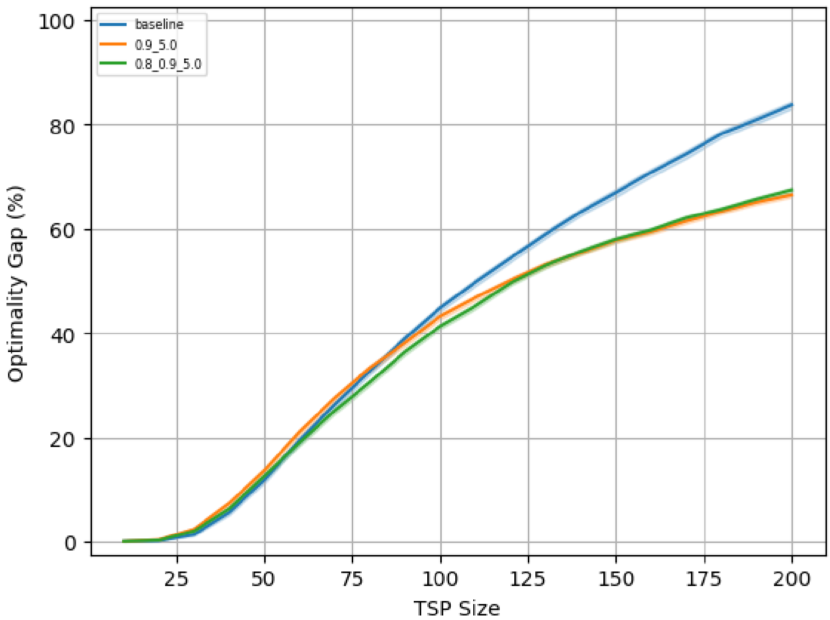


Figure 7.9: Decreasing the model's embedding size and increasing the number of layers enhances the relative improvement obtained by using SizeShiftReg. Models are able to maintain the performance on small sizes while having significant generalization improvements.

To sum up, we have seen that introducing a proper amount of SizeShiftReg improves the generalization performance when training on varying sizes from 20 to 50. However, testing various scenarios that could directly influence how much improvement we could get requires extensive analysis out of the scope of this thesis. Instead, we shift our focus to training on TSP50 to analyze the effects of applying SizeShiftReg when training on a fixed graph size.

Training on a fixed graph size

In this section, we analyze the effects of SizeShiftReg when our models are trained on the TSP50 dataset. We start by using 128,000 data samples for the preliminary results followed by an extension to 384,000 training instances. Instead of experimenting with different coarsening ratios as in the previous section, we focus on using $(0, 8, 0.9)$ combined as it showed the best performance in [9].

Same as before, we train the baseline AGCN model with ReLU activation function and L_2 normalization on the node embeddings. We compare the baseline with two models that use SizeShiftReg with different values of λ (1.0 and 5.0). Results are reported in Figure 7.10. We can see that the model with λ set to 1.0 improves both on the training size (slightly) and on larger instances. In fact, the average optimality gap on all test sizes is reduced from 37.478% for the baseline model to 30.052%, leading to an improvement of 7.426%. On the other hand, increasing λ leads to even better performance on larger sizes, but it degrades the performance on sizes 50 and below.

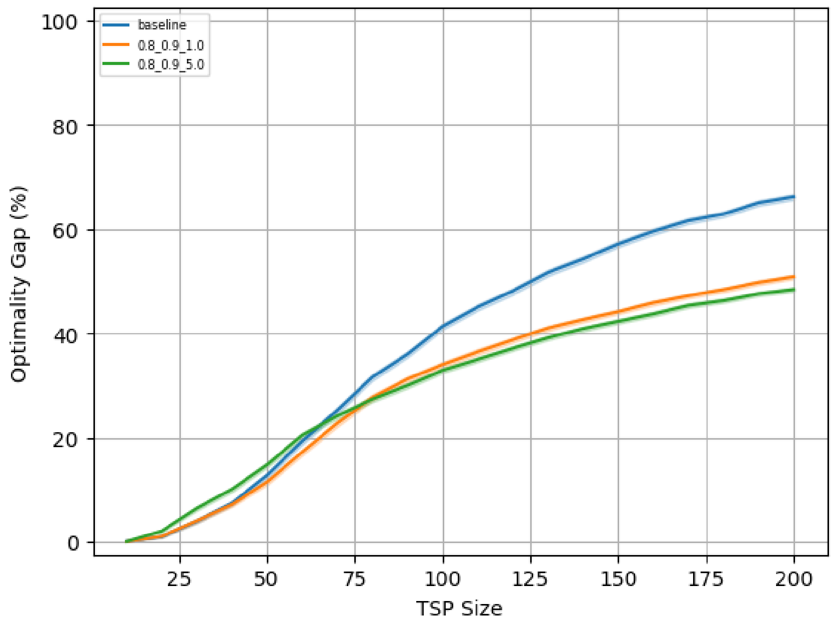


Figure 7.10: Training the model on 128,000 TSP50 instances with SizeShiftReg leads to significant improvements. Introducing a proper amount of SizeShiftReg by setting λ to 1.0 clearly improves the model performance in this setting.

As we find these results very promising, the same analysis is extended to $384{,}000$ TSP50 instances and the results are reported in Figure 7.11. We notice that the previously best-performing λ of 1.0 does not provide the same benefits when we increase the dataset size. In this case, using stronger regularization ($\lambda = 5.0$) leads to better generalization performance, but it also increases the optimality gaps on small graphs. Scaling to more data does not lead to the generalization improvements without training data size performance degradation as we obtained before. However, we report the analysis that shows why these results are valuable and sums up our analysis on the effects of SizeShiftReg.

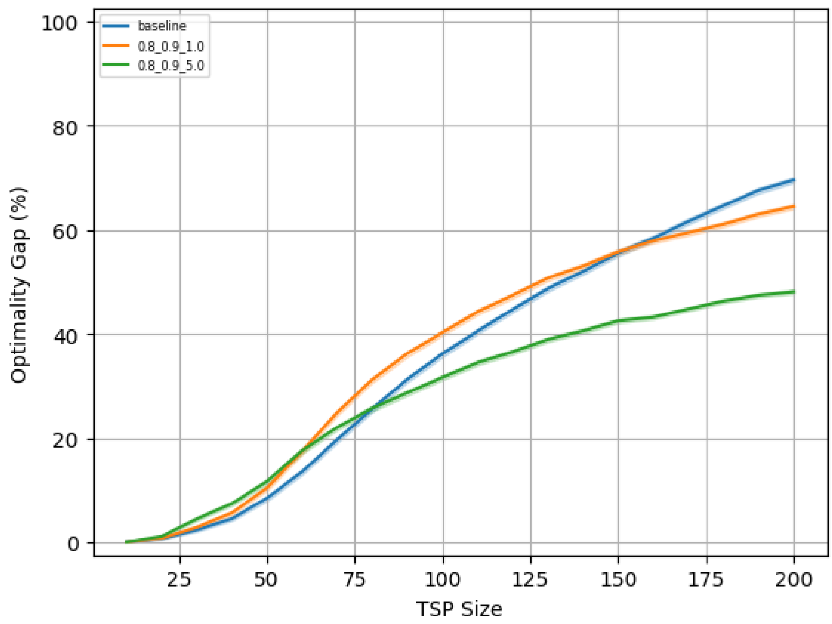
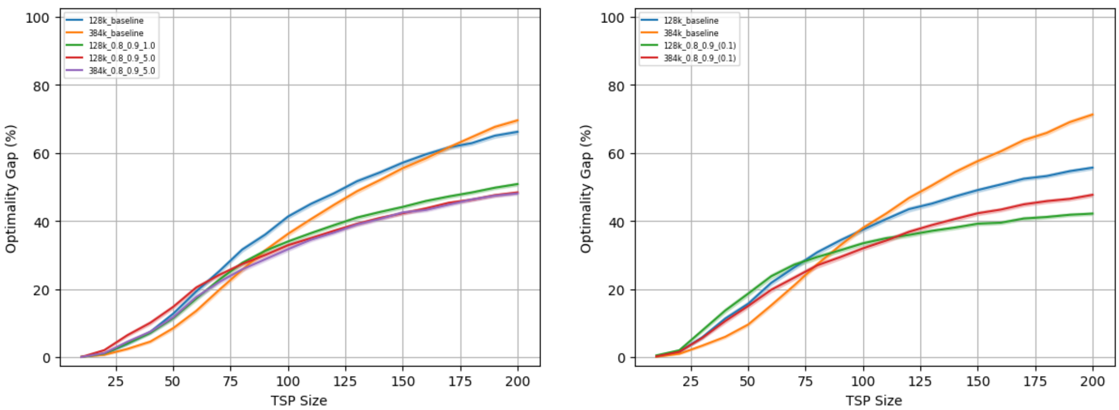


Figure 7.11: Effects of SizeShiftReg when training on $384{,}000$ TSP50 instances. Unfortunately, our improvements do not translate to the larger dataset in a straightforward manner. However, our model is still able to obtain stronger generalization with decreased performance on the training graph sizes. Whether further experimentation could lead to closing this gap on the small sizes while still improving the generalization performance remains a question.

Figure 7.12 sums up the conclusions from our SizeShiftReg analysis. We plot the results of models trained on $128{,}000$ and $384{,}000$ TSP50 instances with prefixes "128k" and "384k", respectively. On the left, we put the results of previously reported analysis on models with ReLU activation function and L_2 node embeddings normalization for both dataset sizes. On the right, we report the same analysis training models with tanh activation function without any normalization as the outputs are already bounded. We will interpret the results of models using ReLU because we have already described their performance, but the models with tanh show similar patterns. We included the tanh results to demonstrate that our insights are relevant across various settings.

Baseline models behave as we already described in Section 7.3.1. That is, adding more data results in model overfitting on the patterns present in the training data size. Then, including SizeShiftReg during training leads to varying generalization improvements. As we already said, SizeShiftReg enhances the performance across sizes leading to a meaningful average optimality gap decrease of 7.4% when training on the lower amount of data. If we increase λ, we get even better generalization, but the model is not able to capture the training data size patterns as well as the baseline. In case of scaling to more data, we were able to significantly improve the generalization performance, but we have not found the proper λ that also keeps the optimality gaps on smaller sizes as good as the baseline. However, if we closely observe the purple line, we notice that the "384k" model with SizeShiftReg has the best generalization performance while capturing the data set patterns as well as any other model besides the one that overfits on the training data size. That is, it combines the benefits of smaller and larger λ when training on smaller amount of data (observe purple, green, and red lines). In fact, it does have the best average optimality gap of 28.668% out of all the presented models. However, whether it can capture the training data patterns as well as the baseline while maintaining similar generalization performance remains an open question. We suspect that this might require higher number of parameters to increase the expressivity combined with stronger SizeShiftReg.



(a) ReLU with L_2 normalization

(b) tanh

Figure 7.12: Summary of SizeShiftReg results.

Training Time Increase

The authors of [9] report the 50% overhead in terms of training time when using SizeShiftReg. In our application, we had to increase the computation time to save memory in certain parts of the implementation because of the huge increase in dataset scale, that we discussed earlier. For this reason (and possibly many others, such as different hardware resources), we observe an average training time increase of 2.69 times with respect to the model without SizeShiftReg. We report the relative differences in training times for one epoch in Table 7.1. Please keep in mind that we possibly use different parameters and resources for different models. Hence, for example, increasing the epoch size 3 times with the same dataset does not lead to a 3 times increase in training time per epoch. However, we report the results of training the models with and without SizeShiftReg within the same setting to ensure a fair comparison. We can see that the overhead gets relatively smaller when increasing the dataset size which is a good indication for further scaling. Moreover, we believe that an optimized implementation paired with better resources can make the overhead closer to the one reported in [9].

Number of Instances	Training Data Size	Epoch Time without SSR	Epoch Time with SSR
128,000	TSP50	7m	20.6m
384,000	TSP20-50	12m	35.7m
384,000	TSP50	28m	60m

Table 7.1: Comparison of training times per epoch with and without SizeShiftReg (SSR).

7.4 Compositional Generalization Results

In this section, we report the results obtained by applying the compositional heatmap procedure [8] to our models. We begin by describing our experimental setting in detail (please refer to Section 7.2 for the preliminary knowledge). Following that, we discuss the trade-offs of using simpler models. Finally, we show that our best model achieves competitive performance compared to Att-GCRN with 9 times fewer parameters.

Experimental Setup

We test three different scales of the AGCN model to analyze the performance/complexity trade-off. Number of parameters varies between 350,000, 850,000, and 1,280,000. Remember, from Section 7.3.2, that we can construct these model scales with different depth/width ratios.

That is, we will first test the models with node and edge embedding sizes of 128, and 4, 10, or 15 GNN encoder layers followed by the models with the embedding size of 96, and 7, 18, or 27 GNN encoder layers. As before, models with 350,000 parameters are trained for 10 epochs while the other two model scales are trained for 20 epochs. Models use either tanh or ReLU activation function which will be specified in the dedicated sections.

Our models are compared against 3 baselines, i.e., a classical solver (Concorde or LKH), heatmaps provided by [8], and our reproduced results of their model (around 11,000,000 parameters). To clarify, we cannot simply report the results from the original paper because they are heavily dependent on the hardware configuration. For this reason, one baseline is obtained by downloading the heatmaps provided by the authors of [8] and running the MCTS on them. Instead, the second baseline uses the trained Att-GCRN provided by the authors to generate the heatmaps. We run the MCTS procedure in exactly the same environment with the same hyperparameters in all our experiments to ensure fair comparison. The authors measure the cumulative running time on all the instances for both heatmap generation and MCTS. Instead, we report only the cumulative time to generate the heatmaps. As we have reused their MCTS procedure, the running time will be exactly the same, and we want to emphasize the models comparison. MCTS is fixed to search for 20, 50, 100, and 1000 seconds on TSP200, TSP500, TSP1000, and TSP10000, respectively. These time constraints are adopted from [31, 6, 7, 29].

Performance results are reported in Table 7.2 and Table 7.3. The results are separated in two tables with the same structure to provide a clear interpretation based on the models trained. For each model, we report the average solution length, optimality gap, and cumulative heatmap generation time on all test instances. We do not report the results of the "Att-GCRN (original model)" baseline for the TSP10000 dataset because we were not able to reproduce the heatmaps due to the GPU memory exception. Moreover, classical solver running times are not reported because we are interested in comparing ML approaches. For the "Att-GCRN (original heatmaps)" baseline we copy the times from the paper as we do not generate the heatmaps ourselves. In our setting, generating heatmaps with their model took way more time than originally reported. We will comment on this shortly. For the model names, we follow the same rule as before, the number after "AGCN" denotes the number of GNN encoder layers while the node (and edge) embedding size comes after the the underscore. For example, "AGCN4_128" denotes the model with 4 layers and embedding size of 128. The flags in the parentheses next to the model indicate whether it masks the edges and/or uses tanh instead of ReLU. Masking the edges means setting the probability of the edges that were removed in graph sparsification to zero as explained in Section 6.2.1. Regarding the model training times, we report them in Table 7.4. Training time for the Att-GCRN model is taken from [8] because we never retrained it in our setting. We only used the available trained model to generate the heatmaps as previously explained.

Finally, before we interpret the results, we should state some important considerations regarding the comparison. First of all, there are no established metrics in the literature to compare optimality gaps relative to model complexity. We will just separately compare optimality gaps differences and discuss the trade-off with respect to the number of parameters. Secondly, as you can see in the tables, there is a significant performance difference between the original heatmaps provided by the authors and our reproduced heatmaps for their model. Again, we did not retrain the model, we just downloaded it and generated the heatmaps. For this reason, we will compare to both of the baselines even though they refer to the same model. Finally, we have a consideration regarding the time needed to generate the heatmaps. We were not able to use as large batch size as they did for their model (again, for memory reasons), so our reproduced heatmap generation times for their model are way larger than in the original paper. Fair time comparison is a major problem in the field, and this is not tied to our case. Hence, please consider time comparisons as the estimates. It is clear that our smaller models are significantly faster, but it is way harder to quantify exactly how much.

Shallow Models Results

In this section, we report the results of the models with embedding size set to 128. We call them shallow models just to distinguish from the subsequent analysis of models with different depth/width ratio. Models use either ReLU or tanh depending on the performance in preliminary beam search analysis. That is, we choose the better performing activation function on each scale. Moreover, we test the masked and non-masked version of the heatmap for every model. Results are reported in Table 7.2.

The first thing we notice across all models is the negligible effect of heatmap masking. In all cases, the difference between the masked and non-masked version is very low. More importantly, there is no pattern where one of the options outperforms the other consistently. This indicates that the differences are likely due to the variation in different runs rather than the method itself. It seems that the models do not assign high probabilities to the edges removed during graph sparsification which is the desired behavior in the learning process. For these reasons, we always consider the masked version for performance comparison.

Looking at the results, we see that none of our models have been able to reach the performance of Att-GCRN. However, remember that our models are approximately 31, 13, and 9 smaller in terms of the number of parameters than the baseline. This is evident in heatmap generation times as well. Our cumulative heatmap generation times are significantly shorter, especially when compared to times where we reproduce the heatmaps in the same setting. As we noted earlier, inference time comparisons are not reliable. Therefore, we will not provide specific quantification, but the results clearly indicate significant improvements.

Our models show an expected pattern of better performance with increased complexity. However, the rate of increase does diminish as we can see that the AGCN10_128 model relatively improves over AGCN4_128 more than the AGCN15_128 over AGCN10_128. That being said, the relative increase in number of parameters is also way larger in the first case. We cannot simply choose the best model scale as it depends on the desirable trade-off for the specific application. Moreover, besides considering the performance/complexity trade-off, smaller models can also extend to additional techniques that require more forward passes. For example, they could allow us to use the extended compositional construction mentioned in Section 6.1.1.

Finally, we analyze how our largest model compares to the baseline. The relative optimality gap differences with respect to the original heatmaps are 0.1613, 0.4275, 0.4418, and 0.75. If we compare to the reproduced heatmaps by running the provided model, we get 0.1112, 0.1734, 0.1597, and NA for TSP10000 as we were not able to reproduce the results. The baseline is better across sizes, even if by a small margin, thus we can be certain that it is indeed a better-performing model. However, significant memory and time savings offered by our model could be more important depending on the application.

Method	TSP200			TSP500			TSP1000			TSP10000		
	Length	Gap (%)	Time	Length	Gap (%)	Time	Length	Gap (%)	Time	Length	Gap (%)	Time
Solver	10.7280	0.0000	/	16.5836	0.0000	/	23.2268	0.0000	/	71.7778	/	/
Att-GCRN (original heatmaps)	10.7470	0.1768	20.62s	16.7902	1.2462	31.17s	23.6274	1.7247	43.94s	74.2262	3.4111	249.6s
Att-GCRN (original model)	10.7524	0.2269	50.91s	16.8323	1.5003	125.85s	23.6927	2.0068	268.85s	/	/	/
AGCN4_128 (ReLU; masked)	10.8091	0.7556	3.53s	16.9699	2.3301	8.02s	23.8839	2.8298	19.69s	75.2402	4.8237	135.99s
AGCN4_128 (ReLU; not masked)	10.8120	0.7895	3.53s	16.9613	2.2778	8.02s	23.8843	2.8315	19.69s	75.2706	4.8662	135.99s
AGCN10_128 (tanh; masked)	10.7699	0.3893	5.39s	16.8767	1.7677	13.15s	23.7749	2.3605	29.24s	74.8187	4.2366	149.22s
AGCN10_128 (tanh; not masked)	10.7717	0.4064	5.39s	16.8846	1.8156	13.15s	23.7521	2.2619	29.24s	74.8093	4.2234	149.22s
AGCN15_128 (tanh; masked)	10.7644	0.3381	7.79s	16.8611	1.6737	18.39s	23.7298	2.1665	38.85s	74.7645	4.1611	164.82s
AGCN15_128 (tanh; not masked)	10.7618	0.3145	7.79s	16.8628	1.6838	18.39s	23.7366	2.1957	38.85s	74.7413	4.1287	164.82s

Table 7.2: Compositional generalization results 1.

Deep Models Results

In this section, we report the results of models with the embedding size set to 96 and 7, 18, or 27 GNN encoder layers. As we have seen in Section 7.3.2, deeper models usually slightly improve compared to their shallow counterparts on the same scale. We test them using the compositional generalization to analyze how they compare in this setting. Moreover, we perform an additional analysis by setting the subgraph sampling size to 75 for our model. In this way, we achieve comparable results to the baseline with 9 times decrease in the number of parameters. To be clear, we remind you that the subgraph sampling size is set to 50 in all the other models (both ours and baselines) as explained in Section 7.2.

As the choice between masking and not masking the heatmaps did not have any influence in previous analysis, we decided to test only the masked heatmaps with deep models. Moreover, we always set the activation function to tanh because it was used for our depth/width analysis in Section 7.3.2. As a consequence, the corresponding flags are removed from the model names in the table to ease the notation. On the other hand, we add an additional flag for the last two models to denote that they sample the graphs of size 75 instead of 50 when constructing the compositional heatmap. Results are reported in Table 7.3. Please note that the results of three baselines are copied for reference.

Deeper models show variable performance in terms of comparison with their shallower counterparts. That is, the deeper model on the smallest scale is worse than the shallow one across all sizes. We remind you that it had weaker performance in direct generalization as reported in Section 7.3.2, so this result was expected. On the other hand, for the other two scales, we have improvements on TSP200, TSP500, and TSP1000, but the performance is slightly worse on

TSP10000. Thus, we were able to close the gap a bit more with respect to the baseline on most sizes, but the deeper models also require more time to generate the heatmaps. The memory requirements are the same because we do not change the number of parameters within the same model scale.

Finally, we perform the experiment with increased subgraph sampling size where we obtain our best results. We set the subgraph sampling size to 75 instead of 50 for our best model (AGCN27_96) and the original Att-GCRN provided by the authors to generate new heatmaps. The Att-GCRN baseline model obtains larger optimality gaps on every size when the subgraph sampling size is increased. On the other hand, our best model (AGCN27_96) improves the performance on all sizes when we increase the subgraph size. As we discussed in Section 6.1, having a model that generalizes well across sizes can be combined with the compositional heatmap approach. This result is one of such examples. Moreover, if we compare it to the strongest baseline (their original heatmaps with subgraphs of 50), we have competitive results. That is, our model is slightly better on TSP200 while it still performs worse on larger sizes with the absolute differences of 0.2745, 0.2735, and 0.6018 on TSP500, TSP1000, and TSP10000, respectively. However, if we compare it to our reproduced results of the baseline Att-GCRN (with its best setting, i.e., subgraph size set to 50), we have basically the same performance. Actually, our model is slightly better on TSP200 and TSP1000, slightly worse on TSP500, while the TSP10000 is not comparable because we were not able to reproduce the results of the original paper due to memory constraints.

To sum up, our final results show that:

- We are able to achieve competitive performance to the Att-GCRN model from [8] with approximately 9 times fewer parameters.
- There is potential in combining simpler models that have improved generalization performance with the compositional heatmap approach.

Regarding the first point, we want to note that the term "competitive" is intentionally loosely-defined. That is, we perform a little bit better than what we were able to reproduce from the paper, but still not as good as rerunning their heatmaps. These comparability issues are ubiquitous in the field, so we cannot always provide a precise quantification. However, we still believe that our results indicate the advantages of using simpler models. The second point is a proof of concept for the direction of future work. That is, we show the potential benefits of combining a simpler model that has improved generalization outside the training size with the compositional construction of heatmaps. Future work can focus on implementing some of the ideas presented in Section 6.1 that directly build upon these results.

Method	TSP200			TSP500			TSP1000			TSP10000		
	Length	Gap (%)	Time	Length	Gap (%)	Time	Length	Gap (%)	Time	Length	Gap (%)	Time
Solver	10.7280	0.0000	/	16.5836	0.0000	/	23.2268	0.0000	/	71.7778	/	/
Att-GCRN (original heatmaps)	10.7470	0.1768	20.62s	16.7902	1.2462	31.17s	23.6274	1.7247	43.94s	74.2262	3.4111	249.6s
Att-GCRN (original model)	10.7524	0.2269	50.91s	16.8323	1.5003	125.85s	23.6927	2.0068	268.85s	/	/	/
AGCN7_96	10.8221	0.8761	4.6s	17.0004	2.5132	9.69s	23.9295	3.0264	20.55s	75.4148	5.0069	135.13s
AGCN18_96	10.7669	0.3622	7.89s	16.8674	1.7129	16.73s	23.7596	2.2946	36.13s	74.8367	4.2617	158.36s
AGCN27_96	10.7570	0.2697	9.01s	16.8391	1.5410	23.06s	23.7204	2.1259	47.37s	74.8349	4.2591	172.58s
Att-GCRN (original model; 75 subgraph size)	10.7625	0.3202	70.94s	16.8617	1.6771	174.47s	23.7348	2.1874	333.82s	/	/	/
AGCN27_96 (75 subgraph size)	10.7453	0.1599	13.09s	16.8358	1.5207	32.38s	23.6884	1.9882	66.39s	74.6582	4.0129	196.28s

Table 7.3: Compositional generalization results 2.

Method	Training Data Sizes	Training Times
Att-GCRN	TSP50	25h
AGCN4_128	TSP20-50	4h
AGCN10_128	TSP20-50	12h 30m
AGCN15_128	TSP20-50	16h
AGCN7_96	TSP20-50	4h 30m
AGCN18_96	TSP20-50	14h 45m
AGCN27_96	TSP20-50	20h

Table 7.4: Compositional generalization model training times.

8
Conclusion

In this thesis, we explored the size generalization of learning-based approaches for solving the TSP, focusing specifically on GNN methods. In particular, we identified and addressed the gaps in current NAR approaches focusing on zero-shot generalization scenario. We begin by extending the analysis of direct NAR generalization through experimenting with simple architectural changes, i.e., model scale, normalization techniques, activation functions, and depth/width ratio. Our results show the positive influence of bounding the scale of node embeddings. However, the results are still far from the desirable generalization performance. Hence, we use the insight obtained from this preliminary analysis to guide our application of SizeShiftReg [9]. SizeShiftReg is a regularization method originally developed for the graph classification task, and this is, to the best of our knowledge, the first application of this technique to the CO domain. We obtain up to 7.4% reduction in the average optimality gap on the test dataset while improving (or at least maintaining) the performance on every graph size. However, we were not able to scale our SizeShiftReg application to the amount of data used in state-of-the-approaches due to the lack of adequate resources. Anyway, our results show the potential to improve direct NAR generalization using carefully designed regularization techniques. Moreover, even without including SizeShiftReg, our model achieves competitive performance compared to the Att-GCRN model [8], one of the top-performing models in the literature, in the compositional generalization scenario, while having 9 times less parameters. Finally, our results demonstrate the advantages of combining simpler models that offer improved generalization performance and the compositional construction of heatmaps. That is, the performance of our model improves with the increase of subgraph sampling size at inference time, while the performance of Att-GCRN baseline model degrades in the same scenario.

Future Work

We identify two avenues for future work. The first one is a direct extension of the work done in this thesis while the second direction considers insight drawn from the concurrent advancements in the literature.

Even though we obtained significant improvements by applying SizeShiftReg, resource constraints have not allowed us to experiment with the amount of data used in most of the state-of-the-art approaches. Further experimentation is required to understand how much improvement can be obtained by applying SizeShiftReg to the CO domain. Furthermore, the application of SizeShiftReg is in no way limited to the specific setting that we considered. For example, with minimal code changes, our implementation can be translated to the AR version of the AGCN architecture. As the recent AR architecture [24] directly generalizes to graphs of sizes up to 1000 nodes, it would be interesting to test the effects of SizeShiftReg in this setting. Regarding the second major focus of our work, we showed that simpler models with improved generalization performance open up possibilities to extend the compositional heatmap procedure. However, we only scratched the surface, and the direct extension could include further experimentation with increased size of sampled subgraphs paired with the two levels of compositional heatmap construction and/or sampling of variable subgraph sizes to identify promising edges across increasing scales.

Regarding the second line of future work, we also consider the latest advancements in the literature. Concurrently with our work, the authors of [56] criticize current heatmap-based approaches for the TSP. In particular, instead of using a GNN to produce the heatmaps, they introduce a simple model with only one tunable parameter τ. It works by applying the softmax activation function on the distance matrix of a TSP instance to obtain the score for each edge. The smoothness of the scores is controlled by the parameter τ. Instead of the gradient descent, they optimize τ with a simple grid search procedure. Their simple method outperforms most of the existing neural solvers in the literature.

From one point of view, their work addresses the same gap that we noticed. That is, current architectures are too complex and competitive performance can be obtained with simpler models. On the other hand, their work completely shifts the focus from typical ML approaches and points out their shortcomings. Moreover, the authors advocate for developing more theoretically sound heatmap generation methods because the current approaches perform worse than existing heuristics. The conclusion that ML approaches have not reached the performance of heuristic solvers is well-known in the literature, but the authors of [56] further analyze this issue.

On the other hand, the authors of [57] point out a fundamental drawback of all current learning-based approaches in the CO domain. That is, the models focus solely on learning from the data while completely ignoring the underlying algorithmic structure of the problem. They propose the neural algorithmic reasoning paradigm to inject algorithmic knowledge into the learning-based CO solvers. In particular, they pretrain the network to execute related algorithms that run in polynomial time (e.g., algorithm for finding the minimum spanning tree in case of the TSP). Their experiments show a significant increase of performance with respect to the networks unaware of algorithmic knowledge. That being said, leveraging algorithmic knowledge has not been explored prior to this work, which opens up a new avenue of future research. We believe that the introduction of algorithmic knowledge into the network paired with a carefully designed regularization technique to ensure robustness against size distribution shifts (such as SizeShiftReg) presents a promising direction towards closing the gap with respect to heuristic approaches.

References

[1] Q. Cappart, D. Chételat, E. B. Khalil, A. Lodi, C. Morris, and P. Veličković, "Combinatorial optimization and reasoning with graph neural networks," *Journal of Machine Learning Research*, vol. 24, no. 130, pp. 1–61, 2023.

[2] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *European Journal of Operational Research*, vol. 290, no. 2, pp. 405–421, 2021.

[3] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, 1972, pp. 85–103.

[4] C. K. Joshi, Q. Cappart, L.-M. Rousseau, and T. Laurent, "Learning the travelling salesperson problem requires rethinking generalization," *Constraints*, vol. 27, no. 1, pp. 70–98, 2022.

[5] Y. Jin and J. F. JáJá, "Network summarization with preserved spectral properties," *arXiv preprint arXiv:1802.04447*, 2018.

[6] Z. Sun and Y. Yang, "Difusco: Graph-based diffusion solvers for combinatorial optimization," *Advances in Neural Information Processing Systems*, vol. 36, pp. 3706–3731, 2023.

[7] R. Qiu, Z. Sun, and Y. Yang, "Dimes: A differentiable meta solver for combinatorial optimization problems," *Advances in Neural Information Processing Systems*, vol. 35, pp. 25 531–25 546, 2022.

[8] Z.-H. Fu, K.-B. Qiu, and H. Zha, "Generalize a small pre-trained model to arbitrarily large tsp instances," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 8, 2021, pp. 7474–7482.

[9] D. Buffelli, P. Liò, and F. Vandin, "Sizeshiftreg: a regularization method for improving size-generalization in graph neural networks," *Advances in Neural Information Processing Systems*, vol. 35, pp. 31 871–31 885, 2022.

[10] L. Pasa, N. Navarin, and A. Sperduti, "Deep learning for graph-structured data," in *HANDBOOK ON COMPUTER LEARNING AND INTELLIGENCE: Volume 2: Deep Learning, Intelligent Control and Evolutionary Computation*. World Scientific, 2022, pp. 585–617.

[11] W. L. Hamilton, *Graph representation learning*. Morgan & Claypool Publishers, 2020.

[12] P. D. Hoff, A. E. Raftery, and M. S. Handcock, "Latent space approaches to social network analysis," *Journal of the american Statistical association*, vol. 97, no. 460, pp. 1090–1098, 2002.

[13] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[15] B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*, 5th ed. Springer Publishing Company, Incorporated, 2012.

[16] I. Goodfellow, *Deep learning*. MIT press, 2016, vol. 196.

[17] K. Kawaguchi, L. P. Kaelbling, and Y. Bengio, "Generalization in deep learning," *arXiv preprint arXiv:1710.05468*, vol. 1, no. 8, 2017.

[18] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning (still) requires rethinking generalization," *Communications of the ACM*, vol. 64, no. 3, pp. 107–115, 2021.

[19] Y. Wang, Q. Yao, J. Kwok, and L. Ni, "Few-shot learning: A survey," *arXiv preprint arXiv:1904.05046*, vol. 1, 2019.

[20] Y. Xian, B. Schiele, and Z. Akata, "Zero-shot learning-the good, the bad and the ugly," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4582–4591.

[21] W. Wang, V. W. Zheng, H. Yu, and C. Miao, "A survey of zero-shot learning: Settings, methods, and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–37, 2019.

[22] P. Veličković and C. Blundell, "Neural algorithmic reasoning," *Patterns*, vol. 2, no. 7, 2021.

[23] C. K. Joshi, T. Laurent, and X. Bresson, "An efficient graph convolutional network technique for the travelling salesman problem," *arXiv preprint arXiv:1906.01227*, 2019.

[24] F. Luo, X. Lin, F. Liu, Q. Zhang, and Z. Wang, "Neural combinatorial optimization with heavy decoder: Toward large scale generalization," *Advances in Neural Information Processing Systems*, vol. 36, pp. 8845–8864, 2023.

[25] W. Kool, H. van Hoof, J. Gromicho, and M. Welling, "Deep policy dynamic programming for vehicle routing problems," in *International conference on integration of constraint programming, artificial intelligence, and operations research*. Springer, 2022, pp. 190–213.

[26] W. Kool, H. Van Hoof, and M. Welling, "Attention, learn to solve routing problems!" *arXiv preprint arXiv:1803.08475*, 2018.

[27] Y.-D. Kwon, J. Choo, B. Kim, I. Yoon, Y. Gwon, and S. Min, "Pomo: Policy optimization with multiple optima for reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 21 188–21 198, 2020.

[28] X. Pan, Y. Jin, Y. Ding, M. Feng, L. Zhao, L. Song, and J. Bian, "H-tsp: Hierarchically solving the large-scale traveling salesman problem," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 8, 2023, pp. 9345–9353.

[29] H. Cheng, H. Zheng, Y. Cong, W. Jiang, and S. Pu, "Select and optimize: Learning to solve large-scale tsp instances," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2023, pp. 1219–1231.

[30] Y. Li, J. Guo, R. Wang, and J. Yan, "From distribution learning in training to gradient search in testing for combinatorial optimization," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[31] Y. Min, Y. Bai, and C. P. Gomes, "Unsupervised learning for solving the travelling salesman problem," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[32] Y. Min and C. P. Gomes, "On size and hardness generalization in unsupervised learning for the travelling salesman problem," *arXiv preprint arXiv:2403.20212*, 2024.

[33] J. Hare, "Dealing with sparse rewards in reinforcement learning," *arXiv preprint arXiv:1910.09281*, 2019.

[34] P. R. d O Costa, J. Rhuggenaath, Y. Zhang, and A. Akcay, "Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning," in *Asian conference on machine learning*. PMLR, 2020, pp. 465–480.

[35] L. Xin, W. Song, Z. Cao, and J. Zhang, "Neurolkh: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem," *Advances in Neural Information Processing Systems*, vol. 34, pp. 7472–7483, 2021.

[36] B. Hudson, Q. Li, M. Malencia, and A. Prorok, "Graph neural network guided local search for the traveling salesperson problem," *arXiv preprint arXiv:2110.05291*, 2021.

[37] Y. Min and C. Gomes, "Unsupervised learning permutations for tsp using gumbel-sinkhorn operator," in *NeurIPS 2023 Workshop Optimal Transport and Machine Learning*, 2023.

[38] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[39] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," *Advances in neural information processing systems*, vol. 28, 2015.

[40] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural computation*, vol. 1, no. 2, pp. 270–280, 1989.

[41] X. Bresson and T. Laurent, "An experimental study of neural networks for variable graphs," in *International Conference on Learning Representations Workshop*, 2018.

[42] S. Ioffe, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[43] J. Ba, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[44] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2011, pp. 315–323.

[45] D. Bahdanau, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[46] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[47] X.-Y. Liu and Z.-H. Zhou, "The influence of class imbalance on cost-sensitive learning: An empirical study," in *sixth international conference on data mining (ICDM'06)*. IEEE, 2006, pp. 970–974.

[48] A. François, Q. Cappart, and L.-M. Rousseau, "How to evaluate machine learning approaches for combinatorial optimization: Application to the travelling salesman problem," *arXiv preprint arXiv:1909.13121*, 2019.

[49] C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann, "Tudataset: A collection of benchmark datasets for learning with graphs," *arXiv preprint arXiv:2007.08663*, 2020.

[50] W. Zellinger, B. A. Moser, T. Grubinger, E. Lughofer, T. Natschläger, and S. Saminger-Platz, "Robust unsupervised domain adaptation for neural networks via moment alignment," *Information Sciences*, vol. 483, pp. 174–191, 2019.

[51] B. Bevilacqua, Y. Zhou, and B. Ribeiro, "Size-invariant graph representations for graph classification extrapolations," in *International Conference on Machine Learning*. PMLR, 2021, pp. 837–851.

[52] G. Yehudai, E. Fetaya, E. Meirom, G. Chechik, and H. Maron, "On size generalization in graph neural networks," in *Int. Conf. Learning Representations*, 2021.

[53] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[54] D. Applegate, R. Bixby, V. Chvatal, and W. Cook, “Concorde tsp solver,” 2006, accessed: 2024-08-15. [Online]. Available: http://www.math.uwaterloo.ca/tsp/concorde.html

[55] D. L. Applegate, R. E. Bixby, V. Chvátal, W. Cook, D. G. Espinoza, M. Goycoolea, and K. Helsgaun, “Certification of an optimal tsp tour through 85,900 cities,” *Operations Research Letters*, vol. 37, no. 1, pp. 11–15, 2009.

[56] Y. Xia, X. Yang, Z. Liu, Z. Liu, L. Song, and J. Bian, “Position: Rethinking post-hoc search-based neural approaches for solving large-scale traveling salesman problems,” *arXiv preprint arXiv:2406.03503*, 2024.

[57] D. G. Georgiev, D. Numeroso, D. Bacciu, and P. Liò, “Neural algorithmic reasoning for combinatorial optimisation,” in *Learning on Graphs Conference*. PMLR, 2024, pp. 28–1.