Learning TSP Requires Rethinking Generalization

Chaitanya K. Joshi
Institute for Infocomm Research, A*STAR, Singapore

Quentin Cappart
Ecole Polytechnique de Montréal, Canada

Louis-Martin Rousseau
Ecole Polytechnique de Montréal, Canada

Thomas Laurent
Loyola Marymount University, LA, USA

Abstract

End-to-end training of neural network solvers for combinatorial optimization problems such as the Travelling Salesman Problem is intractable and inefficient beyond a few hundreds of nodes. While state-of-the-art Machine Learning approaches perform closely to classical solvers when trained on trivially small sizes, they are unable to generalize the learnt policy to larger instances of practical scales. Towards leveraging transfer learning to solve large-scale TSPs, this paper identifies inductive biases, model architectures and learning algorithms that promote generalization to instances larger than those seen in training. Our controlled experiments provide the first principled investigation into such *zero-shot* generalization, revealing that extrapolating beyond training data requires rethinking the neural combinatorial optimization pipeline, from network layers and learning paradigms to evaluation protocols.

2012 ACM Subject Classification Computing methodologies → Neural networks

Keywords and phrases Combinatorial Optimization, Travelling Salesman Problem, Graph Neural Networks, Deep Learning

Digital Object Identifier 10.4230/LIPIcs.CP.2021.33

Related Version *arXiv Pre-Print*: `https://arxiv.org/abs/2006.07054`

Supplementary Material *Software (Source Code and Dataset)*: `https://github.com/chaitjo/learning-tsp`; archived at `swh:1:dir:49220f1be1ae634e106f41948968734ac1569dbd`

Acknowledgements We would like to thank X. Bresson, V. Dwivedi, A. Ferber, E. Khalil, W. Kool, R. Levie, A. Prouvost, P. Veličković and the anonymous reviewers for helpful discussions.

1 Introduction

NP-hard combinatorial optimization problems are the family of integer constrained optimization problems which are intractable to solve optimally at large scales. Robust approximation algorithms to popular problems have immense practical applications and are the backbone of modern industries. Among combinatorial problems, the 2D Euclidean Travelling Salesman Problem (TSP) has been the most intensely studied NP-hard graph problem in the Operations Research (OR) community, with applications in logistics, genetics and scheduling [31]. TSP is intractable to solve optimally above thousands of nodes for modern computers [2]. In practice, the Concorde TSP solver [1] uses linear programming with carefully handcrafted heuristics to find solutions up to tens of thousands of nodes, but with prohibitive execution

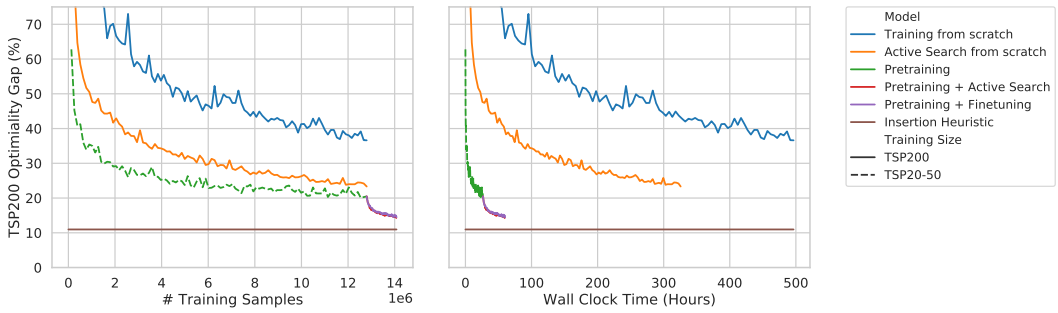


Figure 1 Computational challenges of learning large scale TSP. We compare three identical autoregressive GNN-based models trained on 12.8 Million TSP instances via reinforcement learning. We plot average optimality gap to the Concorde solver on 1,280 held-out TSP200 instances vs. number of training samples (left) and wall clock time (right) during the learning process. Training on large TSP200 from scratch is intractable and sample inefficient. Active Search [4], which learns to directly overfit to the 1,280 held-out samples, further demonstrates the computational challenge of memorizing very few TSP200 instances. Comparatively, learning efficiently from trivial TSP20-TSP50 allows models to better generalize to TSP200 in a zero-shot manner, indicating positive knowledge transfer from small to large graphs. Performance can further improve via rapid finetuning on 1.28 Million TSP200 instances or by Active Search. Within our computational budget, a simple non-learnt *furthest insertion* heuristic still outperforms all models. Precise experimental setup is described in **Appendix A**.

times.[1] Besides, the development of problem-specific OR solvers such as Concorde for novel or under-studied problems arising in scientific discovery [43] or computer architecture [37] requires significant time and specialized knowledge.

An alternate approach by the Machine Learning community is to develop generic learning algorithms which can be trained to solve *any* combinatorial problem directly from problem instances themselves [5]. Using classical problems such as TSP, Minimum Vertex Cover and Boolean Satisfiability as benchmarks, recent *end-to-end* approaches [28, 46, 33] leverage advances in graph representation learning [29, 19] and have shown competitive performance with OR solvers on trivially small problem instances up to few hundreds of nodes. Once trained, approximate solvers based on Graph Neural Networks (GNNs) have significantly favorable time complexity than their OR counterparts, making them highly desirable for real-time decision-making problems such as TSP and the associated class of Vehicle Routing Problems (VRPs).

1.1 Motivation

Scaling end-to-end approaches to practical and real-world instances is still an open question [5] as the training phase of state-of-the-art models on large graphs is extremely time-consuming. For graphs larger than few hundreds of nodes, the gap between GNN-based solvers and simple non-learnt heuristics is especially evident for routing problems like TSP [28, 30].

As an illustration, Figure 1 presents the computational challenge of learning TSP on 200-node graphs (TSP200) in terms of both sample efficiency and wall clock time. Surprisingly, it is difficult to outperform a simple insertion heuristic when directly training on 12.8 Million TSP200 samples for 500 hours on university-scale hardware.

[1] The largest TSP solved by Concorde to date has 109,399 nodes with a total running time of 7.5 months.

We advocate for an alternative to expensive large-scale training: learning efficiently from trivially small TSP and transferring the learnt policy to larger graphs in a *zero-shot* fashion or via fast finetuning. Thus, identifying promising inductive biases, architectures and learning paradigms that enable such zero-shot generalization to large and more complex instances is a key concern for training practical solvers for real-world problems.

1.2 Contributions

Towards end-to-end learning of *scale-invariant* TSP solvers, we unify several state-of-the-art architectures and learning paradigms [40, 30, 12, 27] into one experimental pipeline and provide the first principled investigation on zero-shot generalization to large instances. Our findings suggest that learning scale-invariant TSP solvers requires rethinking the status quo of neural combinatorial optimization to explicitly account for generalization:

- The prevalent evaluation paradigm overshadows models' poor generalization capabilities by measuring performance on fixed or trivially small TSP sizes.
- Generalization performance of GNN aggregation functions and normalization schemes benefits from explicit redesigns which account for shifting graph distributions, and can be further boosted by enforcing regularities such as constant graph diameters when defining problems using graphs.
- Autoregressive decoding enforces a sequential inductive bias which improves generalization over non-autoregressive models, but is costly in terms of inference time.
- Models trained with supervision are more amenable to post-hoc search, while reinforcement learning approaches scale better with more computation as they do not rely on labelled data.

We open-source our framework and datasets[2] to encourage the community to go beyond evaluating performance on fixed TSP sizes, develop more expressive and scale-invariant GNNs, as well as study transfer learning for combinatorial problems.

2 Related Work

Neural Combinatorial Optimization. In a recent survey, Bengio et al. [5] identified three broad approaches to leveraging machine learning for combinatorial optimization problems: learning alongside optimization algorithms [18, 8], learning to configure optimization algorithms [55, 14], and end-to-end learning to approximately solve optimization problems, *a.k.a.* neural combinatorial optimization [53, 4].

State-of-the-art end-to-end approaches for TSP use Graph Neural Networks (GNNs) [29, 19] and *sequence-to-sequence* learning [48] to construct approximate solutions directly from problem instances. Architectures for TSP can be classified as: (1) autoregressive approaches, which build solutions in a step-by-step fashion [28, 12, 30, 35]; and (2) non-autoregressive models, which produce the solution in one shot [40, 39, 27]. Models can be trained to imitate optimal solvers via supervised learning or by minimizing the length of TSP tours via reinforcement learning.

Other classical problems tackled by similar architectures include Vehicle Routing [38, 9], Maximum Cut [28], Minimum Vertex Cover [33], Boolean Satisfiability [46, 62], and Graph Coloring [23]. Using TSP as an illustration, we present a unified pipeline for characterizing neural combinatorial optimization architectures in Section 3.

[2] `https://github.com/chaitjo/learning-tsp`

Notably, TSP has emerged as a challenging testbed for neural combinatorial optimization. Whereas generalization to problem instances larger and more complex than those seen in training has at least partially been demonstrated on non-sequential problems such as SAT, MaxCut, and MVC [28, 33, 46], the same architectures do not show strong generalization for TSP [30, 27].

Combinatorial Optimization and GNNs. From the perspective of graph representation learning, algorithmic and combinatorial problems have recently been used to characterize the expressive power of GNNs [44]. An emerging line of work on learning to execute local graph algorithms [51] has lead to the development of provably more expressive GNNs [10] and improved understanding of their generalization capability [60, 61]. Towards tackling realistic and large-scale combinatorial problems, this paper aims to quantify the limitations of prevalent GNN architectures and learning paradigms via zero-shot generalization to problems larger than those seen during training.

Novel Applications. Advances on classical combinatorial problems have shown promising results in downstream applications to novel or under-studied optimization problems in the physical sciences [20, 47] and computer architecture [36, 41], where the development of exact solvers is expensive and intractable. For example, autoregressive architectures provide a strong inductive bias for device placement optimization problems [37, 65], while non-autoregressive models [7] are competitive with autoregressive approaches [26, 63] for molecule generation tasks.

3 Neural Combinatorial Optimization Pipeline

Many NP-hard problems can be formulated as sequential decision making tasks on graphs due to their highly structured nature. Towards a controlled study of neural combinatorial optimization, we unify recent ideas [40, 30, 12, 27] via a five stage end-to-end pipeline illustrated in Figure 2. Our discussion focuses on the Travelling Salesman Problem (TSP), but the pipeline presented is generic and can be extended to characterize modern architectures for several NP-hard graph problems.

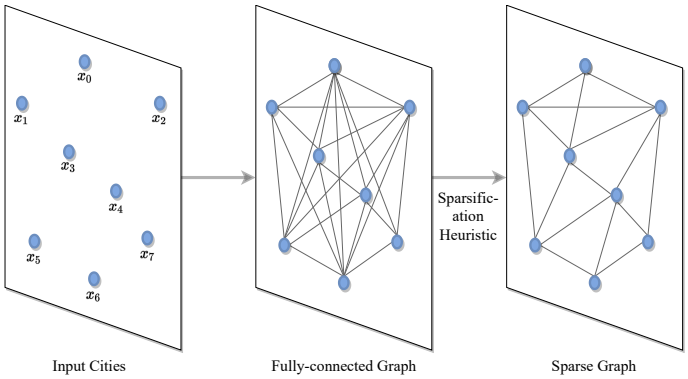
3.1 Problem Definition

The 2D Euclidean TSP is defined as follows: *"Given a set of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"* Formally, given a fully-connected input graph of n cities (nodes) in the two dimensional unit square $S = \{x_i\}_{i=1}^{n}$ where each $x_i \in [0, 1]^2$, we aim to find a permutation of the nodes π, termed a tour, that visits each node once and has the minimum total length, defined as:

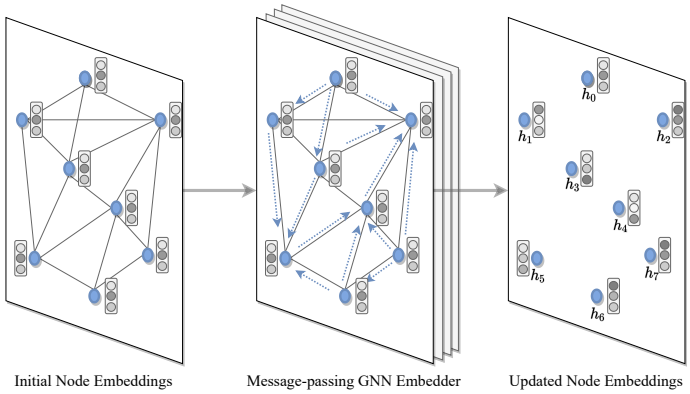
$$L(\pi|s) = \|x_{\pi_n} - x_{\pi_1}\|_2 + \sum_{i=1}^{n-1} \|x_{\pi_i} - x_{\pi_{i+1}}\|_2, \tag{1}$$

where $\|\cdot\|_2$ denotes the ℓ_2 norm.

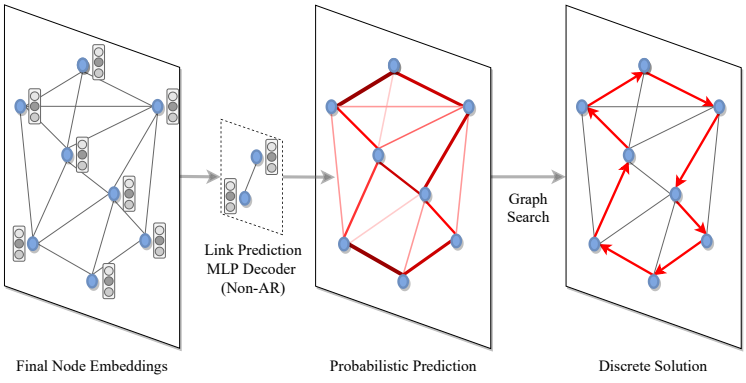
Graph Sparsification. Classically, TSP is defined on fully-connected graphs. Graph sparsification heuristics based on k-nearest neighbors aim to reduce TSP graphs, enabling models to scale up to large instances where pairwise computation for all nodes is intractable [28]



(a) Problem Definition: TSP is formulated via a fully-connected graph of cities/nodes. The graph can be sparsified via heuristics such as k-nearest neighbors.



(b) Graph Embedding: Embeddings for each graph node are obtained using a Graph Neural Network encoder. At each layer, nodes gather features from their neighbors to represent local graph structure via recursive message passing.



(c) Solution Decoding & Search: Probabilities are assigned to each node for belonging to the solution set, either independent of one-another (*i.e.* Non-autoregressive decoding) or conditionally through graph traversal (*i.e.* Autoregressive decoding). The predicted probabilities are converted into discrete decisions through classical graph search techniques such as greedy search or beam search.

Figure 2 End-to-end neural combinatorial optimization pipeline: The entire model in trained end-to-end via imitating an optimal solver (*i.e.* supervised learning) or through minimizing a cost function (*i.e.* reinforcement learning).

or learn faster by reducing the search space [27]. Notably, problem-specific graph reduction techniques have proven effective for out-of-distribution generalization to larger graphs for other NP-hard problems such as MVC and SAT [33].

Fixed size vs. variable size graphs. Most work on learning for TSP has focused on training with a fixed graph size [4, 30], likely due to ease of implementation. Learning from multiple graph sizes naturally enables better generalization within training size ranges, but its impact on generalization to larger TSP instances remains to be analyzed.

3.2 Graph Embedding

A Graph Neural Network (GNN) encoder computes d-dimensional representations for each node in the input TSP graph. At each layer, nodes gather features from their neighbors to represent local graph structure via recursive message passing [19]. Stacking L layers allows the network to build representations from the L-hop neighborhood of each node. Let h_i^ℓ and e_{ij}^ℓ denote respectively the node and edge feature at layer ℓ associated with node i and edge ij. We define the feature at the next layer via an *anisotropic* message passing scheme using an edge gating mechanism [6]:

$$h_i^{\ell+1} = h_i^\ell + \mathrm{ReLU}\Big(\textsc{Norm}\Big(U^\ell h_i^\ell + \textsc{Aggr}_{j\in\mathcal{N}_i}\Big(\sigma(e_{ij}^\ell)\odot V^\ell h_j^\ell\Big)\Big)\Big)\ , \tag{2}$$

$$e_{ij}^{\ell+1} = e_{ij}^\ell + \mathrm{ReLU}\Big(\textsc{Norm}\Big(A^\ell e_{ij}^\ell + B^\ell h_i^\ell + C^\ell h_j^\ell\Big)\Big), \tag{3}$$

where $U^\ell, V^\ell, A^\ell, B^\ell, C^\ell \in \mathbb{R}^{d\times d}$ are learnable parameters, Norm denotes the normalization layer (BatchNorm [25], LayerNorm [3]), Aggr represents the neighborhood aggregation function (Sum, Mean or Max), σ is the sigmoid function, and $\odot$ is the Hadamard product. As inputs $h_i^{\ell=0}$ and $e_{ij}^{\ell=0}$, we use d-dimensional linear projections of the node coordinate x_i and the euclidean distance $\|x_i - x_j\|_2$, respectively.

Anisotropic Neighborhood Aggregation. We make the aggregation function anisotropic or directional via a dense attention mechanism which scales the neighborhood features $h_j, \forall j \in \mathcal{N}_i$, using edge gates $\sigma(e_{ij})$. Anisotropic and attention-based GNNs such as Graph Attention Networks [50] and Gated Graph ConvNets [6] have been shown to outperform isotropic Graph ConvNets [29] across several challenging domains [13], including TSP [30, 27].

3.3 Solution Decoding

Non-autoregressive Decoding (NAR). Consider TSP as a link prediction task: each edge may belong/not belong to the optimal TSP solution independent of one another [40]. We define the edge predictor as a two layer MLP on the node embeddings produced by the final GNN encoder layer L, following Joshi et al. [27]. For adjacent nodes i and j, we compute the unnormalized edge logits:

$$\hat{p}_{ij} = W_2\Big(\mathrm{ReLU}\ \big(W_1\big(\ [\ h_G,\ h_i^L,\ h_j^L\ ]\ \big)\big)\Big), \quad \text{where}\ \ h_G = \frac{1}{n}\sum_{i=0}^{n} h_i^L, \tag{4}$$

$W_1 \in \mathbb{R}^{3d\times d}, W_2 \in \mathbb{R}^{d\times 2}$, and $[\cdot,\cdot,\cdot]$ is the concatenation operator. The logits $\hat{p}_{ij}$ are converted to probabilities over each edge p_{ij} via a softmax.

Autoregressive Decoding (AR). Although NAR decoders are fast as they produce predictions in one shot, they ignore the sequential ordering of TSP tours. Autoregressive decoders, based on attention [12, 30] or recurrent neural networks [53, 35], explicitly model this sequential inductive bias through step-by-step graph traversal.

We follow the attention decoder from Kool et al. [30], which starts from a random node and outputs a probability distribution over its neighbors at each step. Greedy search is used to perform the traversal over n time steps and masking enforces constraints such as not visiting previously visited nodes.

At time step t at node i, the decoder builds a context $\hat{h}_i^C$ for the partial tour $\pi'_{t'}$, generated at time $t' < t$, by packing together the graph embedding h_G and the embeddings of the first and last node in the partial tour: $\hat{h}_i^C = W_C \left[h_G,\ h^L_{\pi'_{t-1}},\ h^L_{\pi'_1} \right]$, where $W_C \in \mathbb{R}^{3d \times d}$ and learned placeholders are used for $h^L_{\pi'_{t-1}}$ and $h^L_{\pi'_1}$ at $t = 1$. The context $\hat{h}_i^C$ is then refined via a standard Multi-Head Attention (MHA) operation [49] over the node embeddings:

$$h_i^C = \text{MHA}\Big(Q = \hat{h}_i^C, K = \{h_1^L, \dots, h_n^L\}, V = \{h_1^L, \dots, h_n^L\}\Big), \tag{5}$$

where Q, K, V are inputs to the M-headed MHA ($M = 8$). The unnormalized logits for each edge e_{ij} are computed via a final attention mechanism between the context h_i^C and the embedding h_j:

$$\hat{p}_{ij} = \begin{cases} C \cdot \tanh\left(\frac{\left(W_Q h_i^C\right)^T \cdot \left(W_K h_j^L\right)}{\sqrt{d}} \right) & \text{if } j \neq \pi_{t'} \quad \forall t' < t \\ -\infty & \text{otherwise.} \end{cases} \tag{6}$$

The tanh is used to maintain the value of the logits within $[-C, C]$ ($C = 10$) [4]. The logits $\hat{p}_{ij}$ at the current node i are converted to probabilities p_{ij} via a softmax over all edges.

Inductive Biases. NAR approaches, which make predictions over edges independently of one-another, have shown strong out-of-distribution generalization for non-sequential problems such as SAT and MVC [33]. On the other hand, AR decoders come with the sequential/tour constraint built-in and are the default choice for routing problems [30]. Although both approaches have shown close to optimal performance on fixed and small TSP sizes under different experimental settings, it is important to fairly compare which inductive biases are most useful for generalization.

3.4 Solution Search

Greedy Search. For AR decoding, the predicted probabilities at node i are used to select the edge to travel along at the current step via sampling from the probability distribution p_i or greedily selecting the most probable edge p_{ij}, *i.e.* greedy search. Since NAR decoders directly output probabilities over all edges independent of one-another, we can obtain valid TSP tours using greedy search to traverse the graph starting from a random node and masking previously visited nodes. Thus, the probability of a partial tour π' can be formulated as $p(\pi') = \prod_{j' \sim i' \in \pi'} p_{i'j'}$, where each node j' follows node i'.

Beam Search and Sampling. During inference, we can increase the capacity of greedy search via limited width breadth-first beam search, which maintains the b most probable tours during decoding. Similarly, we can sample b solutions from the learnt policy and select the shortest tour among them. Naturally, searching longer, with more sophisticated

techniques [16, 58], or sampling more solutions allows trading off run time for solution quality. However, it has been noted that using large b for search/sampling or local search during inference may overshadow an architecture's inability to generalize [15]. To better understand generalization, we focus on using greedy search and beam search/sampling with small $b = 128$.

3.5 Policy Learning

Supervised Learning. Models can be trained end-to-end via imitating an optimal solver at each step (*i.e.* supervised learning). For models with NAR decoders, the edge predictions are linked to the ground-truth TSP tour by minimizing the binary cross-entropy loss for each edge [40, 27]. For AR architectures, at each step, we minimize the cross-entropy loss between the predicted probability distribution over all edges leaving the current node and the next node from the groundtruth tour, following Vinyals et al. [53]. We use teacher-forcing to stabilize training [57].

Reinforcement Learning. Reinforcement learning is a elegant alternative in the absence of groundtruth solutions, as is often the case for understudied combinatorial problems. Models can be trained by minimizing problem-specific cost functions (the tour length in the case of TSP) via policy gradient algorithms [4, 30] or Q-Learning [28]. We focus on policy gradient methods due to their simplicity, and define the loss for an instance s parameterized by the model θ as $\mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)}[L(\pi)]$, the expectation of the tour length $L(\pi)$, where $p_\theta(\pi|s)$ is the probability distribution from which we sample to obtain the tour $\pi|s$. We use the REINFORCE gradient estimator [56] to minimize $\mathcal{L}$:

$$\nabla\mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)}\left[(L(\pi) - b(s))\,\nabla \log p_\theta(\pi|s)\right], \tag{7}$$

where the baseline $b(s)$ reduces gradient variance. Our experiments compare standard critic network baselines [4, 12] and the greedy rollout baseline proposed by Kool et al. [30].

4 Experiments

4.1 Controlled Experiment Setup

We design controlled experiments to probe the unified pipeline described in Section 3 in order to identify inductive biases, architectures and learning paradigms that promote zero-shot generalization. We focus on learning efficiently from small problem instances (TSP20-50) and measure generalization to a wider range of sizes, including large instances which are intractable to learn from (*e.g.* TSP200). We aim to fairly compare state-of-the-art ideas in terms of model capacity and training data, and expect models with good inductive biases for TSP to: (1) learn trivially small TSPs without hundreds of millions of training samples and model parameters; and (2) generalize reasonably well across smaller and larger instances than those seen in training. To quantify "good" generalization, we additionally evaluate our models against a simple, non-learnt *furthest insertion* heuristic baseline [30].

Training Datasets. Our experiments focus on learning from variable TSP20-50 graphs. We also compare to training on fixed graph sizes TSP20, TSP50, TSP100, which have been the default choice in TSP literature. In the supervised learning paradigm, we generate a training set of 1,280,000 TSP samples and groundtruth tours using Concorde. Models are trained using the Adam optimizer for 10 epochs with a batch size of 128 and a fixed learning rate

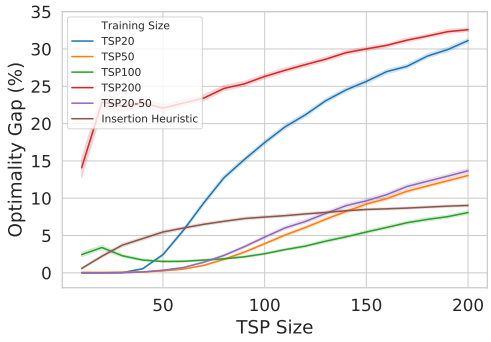


Figure 3 Learning from various TSP sizes. The prevalent protocol of evaluation on training sizes overshadows brittle out-of-distribution performance to larger and smaller graphs.

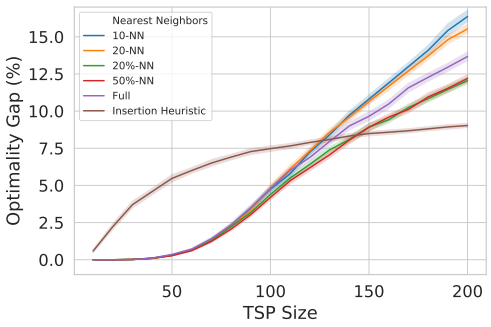


Figure 4 Impact of graph sparsification. Maintaining a constant graph diameter across TSP sizes leads to better generalization on larger problems than using full graphs.

$1e-4$. For reinforcement learning, models are trained for 100 epochs on 128,000 TSP samples which are randomly generated for each epoch (without optimal solutions) with the same batch size and learning rate. Thus, both learning paradigms see 12,800,000 TSP samples in total. Considering that TSP20-50 are trivial in terms of complexity as they can be solved by simpler non-learnt heuristics, training good solvers at this scale should ideally not require millions of instances.

Model Hyperparameters. For models with AR decoders, we use 3 GNN encoder layers followed by the attention decoder head, setting hidden dimension $d = 128$. For NAR models, we use the same hidden dimension and opt for 4 GNN encoder layers followed by the edge predictor. This results in approximately 350,000 trainable parameters for each model, irrespective of decoder type. Unless specified, most experiments use our best model configuration: AR decoding scheme and Graph ConvNet encoder with Max aggregation and BatchNorm (with batch statistics). All models are trained via supervised learning except when comparing learning paradigms.

Evaluation. We compare models on a held-out test set of 25,600 TSP samples, consisting of 1,280 samples each of TSP10, TSP20, ..., TSP200. Our evaluation metric is the optimality gap *w.r.t.* the Concorde solver, *i.e.* the average percentage ratio of predicted tour lengths relative to optimal tour lengths. To compare design choices among identical models, we plot line graphs of the optimality gap as TSP size increases (along with a 99%-ile confidence interval) using beam search with a width of 128. Compared to previous work which evaluated models on fixed problem sizes [4, 12, 30], our evaluation protocol identifies not only those models that perform well on training sizes, but also those that generalize better than non-learnt heuristics for large instances which are intractable to train on.

4.2 Does learning from variable graphs help generalization?

We train five identical models on fully connected graphs of instances from TSP20, TSP50, TSP100, TSP200 and variable TSP20-50. The line plots of optimality gap across TSP sizes in Figure 3 indicates that learning from variable TSP sizes helps models retain performance across the range of graph sizes seen during training (TSP20-50). Variable graph training

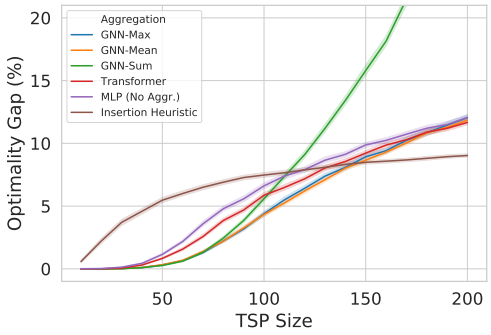


Figure 5 Impact of GNN aggregation functions. For larger graphs, aggregation functions that are agnostic to node degree (MEAN and MAX) are able to outperform theoretically more expressive aggregators.

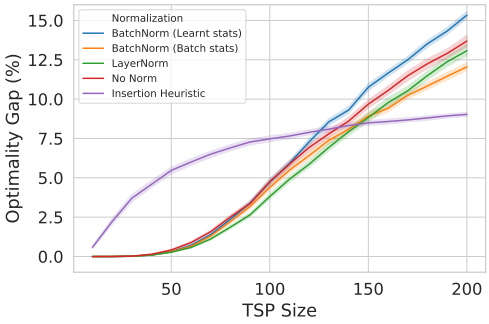


Figure 6 Impact of normalization schemes. Modifying BatchNorm to account for changing graph statistics leads to better generalization on larger graphs.

compared to training solely on the maximum sized instances (TSP50) leads to marginal gains on small instances but, somewhat counter-intuitively, does not enable better generalization to larger problems. Learning from small TSP20 is unable to generalize to large sizes while TSP100 models generalize poorly to trivially easy sizes, suggesting that the prevalent protocol of evaluation on training sizes [30, 27] overshadows brittle out-of-distribution performance.

Training on TSP200 graphs is intractable within our computational budget, see Figure 1. TSP100 is the only model which generalizes better to large TSP200 than the non-learnt baseline. However, training on TSP100 can also be prohibitively expensive: one epoch takes approximately 8 hours (TSP100) vs. 2 hours (TSP20-50) (details in Appendix B). For rapid experimentation, we train efficiently on variable TSP20-50 for the rest of our study.

4.3 What is the best graph sparsification heuristic?

Figure 4 compares full graph training to the following heuristics: (1) **Fixed node degree** across graph sizes, via connecting each node in TSPn to its k-nearest neighbors, enabling GNNs layers to specialize to constant degree k; and (2) **Fixed graph diameter** across graph sizes, via connecting each node in TSPn to its $n \times k\%$-nearest neighbors, ensuring that the same number of message passing steps are required to diffuse information across both small and large graphs.

Although both sparsification techniques lead to faster convergence on training instance sizes, we find that only approach (2) leads to better generalization on larger problems than using full graphs. Consequently, all further experiments use approach (2) to operate on sparse 20%-nearest neighbors graphs. Our results also suggest that developing more principled graph reduction techniques beyond simple k-nearest neighbors for augmenting learning-based approaches may be a promising direction.

4.4 What is the relationship between GNN aggregation functions and normalization layers?

In Figure 5, we compare identical models with anisotropic SUM, MEAN and MAX aggregation functions. As baselines, we consider the Transformer encoder on full graphs [12, 30] as well as a structure-agnostic MLP on each node, which can be instantiated by not using any aggregation function in Eq.(2), *i.e.* $h_i^{\ell+1} = h_i^{\ell} + \text{ReLU}\left(\text{NORM}\left(U^{\ell} h_i^{\ell}\right)\right)$.

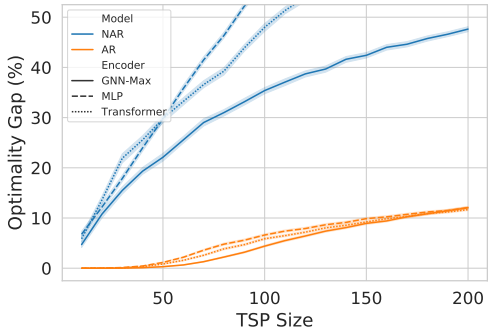


Figure 7 Comparing AR and NAR decoders. Sequential decoding is a powerful inductive bias for TSP as it enables significantly better generalization, even in the absence of graph structure (MLP encoders).

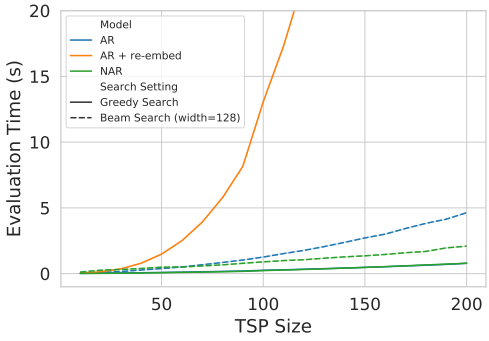


Figure 8 Inference time for various decoders. One-shot NAR decoding is significantly faster than sequential AR, especially when re-embedding the graph at each decoding step [28].

We find that the choice of GNN aggregation function does not have an impact when evaluating models within the training size range TSP20-50. As we tackle larger graphs, GNNs with aggregation functions that are agnostic to node degree (Mean and Max) are able to outperform Transformers and MLPs. Importantly, the theoretically more expressive Sum aggregator [59] generalizes worse than structure-agnostic MLPs, as it cannot handle the distribution shift in node degree and neighborhood statistics across graph sizes, leading to unstable or exploding node embeddings [51]. We use the Max aggregator in further experiments, as it generalizes well for both AR and NAR decoders (not shown).

We also experiment with the following normalization schemes: (1) standard BatchNorm which learns mean and variance from training data, as well as (2) BatchNorm with batch statistics; and (3) LayerNorm, which normalizes at the embedding dimension instead of across the batch. Figure 6 indicates that BatchNorm with batch statistics and LayerNorm are able to better account for changing statistics across different graph sizes. Standard BatchNorm generalizes worse than not doing any normalization, thus our other experiments use BatchNorm with batch statistics.

Poor performance on large graphs than those seen during training can be explained by unstable node and graph-level representations due to the choice of aggregation and normalization schemes. Using Max aggregators and BatchNorm with batch statistics are temporary hacks to overcome the failure of the current architectural components. Overall, our results suggest that inference beyond training sizes will require the development of expressive GNN mechanisms that are able to leverage global graph topology [17, 52] while being invariant to distribution shift [32].

4.5 Which decoder has a better inductive bias for TSP?

Figure 7 compares NAR and AR decoders for identical models. To isolate the impact of the decoder's inductive bias without the inductive bias imposed by GNNs, we also show Transformer encoders on full graphs as well as structure-agnostic MLPs. Within our experimental setup, AR decoders are able to fit the training data as well as generalize significantly better than NAR decoders, indicating that sequential decoding is powerful for TSP even without graph information.

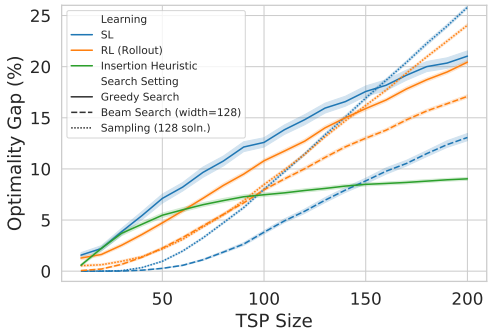


Figure 9 Comparing solution search settings. Under greedy decoding, RL demonstrates better performance and generalization. Conversely, SL models improve over their RL counterparts when performing beam search or sampling.

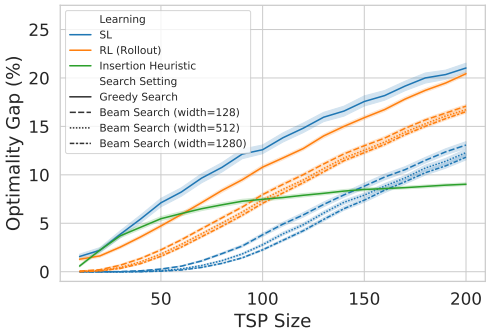


Figure 10 Impact of increasing beam width. Teacher-forcing during SL leads to poor generalization under greedy decoding, but makes the probability distribution more amenable to beam search.

Conversely, NAR architectures are a poor inductive bias as they require significantly more computation to perform competitively to AR decoders. For instance, recent work [40, 27] used more than 30 layers with over 10 Million parameters. We believe that such overparameterized networks are able to memorize all patterns for small TSP training sizes [64], but the learnt policy is unable to generalize beyond training graph sizes. At the same time, when compared fairly within the same experimental settings, NAR decoders are significantly faster than AR decoders described in Section 3.3 as well as those which re-embed the graph at each decoding step [28], see Figure 8.

4.6 How does the learning paradigms impact the search phase?

Identical models are trained via supervised learning (SL) and reinforcement learning (RL) (We show only the greedy rollout baseline for clarity.). Figure 9 illustrates that, when using greedy decoding during inference, RL models perform better on the training size as well as on larger graphs. Conversely, SL models improve over their RL counterparts when performing beam search or sampling.

In Appendix D, we find that the rollout baseline, which encourages better greedy behaviour, leads to the model making very confident predictions about selecting the next node at each decoding step, even out of training size range. In contrast, SL models are trained with teacher forcing, *i.e.* imitating the optimal solver at each step instead of using their own prediction. This results in less confident predictions and poor greedy decoding, but makes the probability distribution more amenable to beam search and sampling, as shown in Figure 10. Our results advocate for tighter coupling between the training and inference phase of learning-driven TSP solvers, mirroring recent findings in generative models for text [21].

4.7 Which learning paradigm scales better?

Our experiments till this point have focused on isolating the impact of various pipeline components on zero-shot generalization under limited computation. At the same time, recent results on natural language have highlighted the power of large scale pre-training for effective transfer learning [42]. To better understand the impact of learning paradigms when scaling

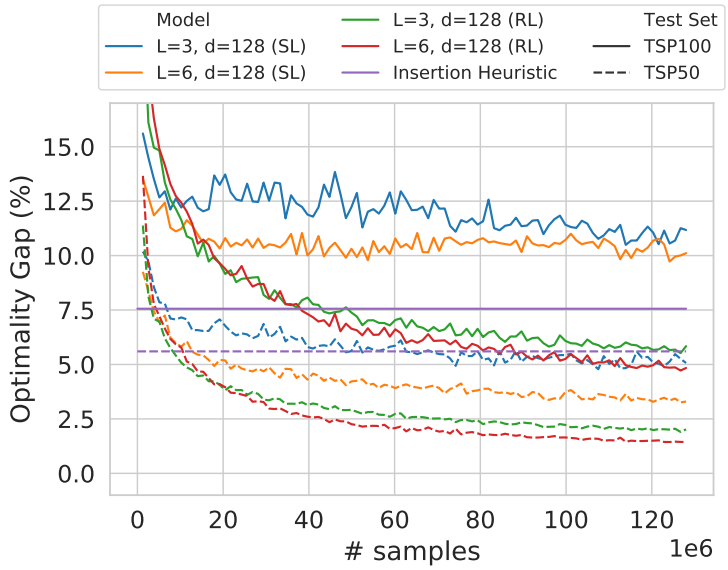


Figure 11 Scaling computation and parameters for SL and RL-trained models. All models are trained on TSP20-50. We plot optimality gap on 1,280 held-out samples of both TSP50 (performance on training size) and TSP100 (out-of-distribution generalization) under greedy decoding. Note that SL models are less amenable than RL models to greedy search. RL models are able to keep improving their performance within as well as outside of training size range with more data. On the other hand, SL performance is bottlenecked by the need for optimal groundtruth solutions.

computation, we double the model parameters (up to 750,000) and train on tens times more data (12.8M samples) for AR architectures. We monitor optimality gap on the training size range (TSP20-50) as well as a larger size (TSP100) vs. the number of training samples.

In Figure 11, we see that increasing model capacity leads to better learning. Notably, RL models, which train on unique randomly generated samples throughout, are able to keep improving their performance within as well as outside of training size range as they see more samples. On the other hand, SL is bottlenecked by the need for optimal groundtruth solutions: SL models iterate over the same 1.28M unique labelled samples and stop improving at a point. Beyond favorable inductive biases, distributed and sample-efficient RL algorithms [45] may be a key ingredient for learning from larger TSPs beyond tens of nodes.

5 Conclusion

Learning-driven solvers for combinatorial problems such as the Travelling Salesman Problem have shown promising results for trivially small instances up to a few hundred nodes. However, scaling such *end-to-end* learning approaches to real-world instances is still an open question as training on large graphs is extremely time-consuming. As a motivating example, we have demonstrated that state-of-the-art techniques are unable to outperform simple insertion heuristics on TSP beyond 200 nodes when trained on university-scale hardware.

This paper advocates for an alternative to expensive large-scale training: the generalization gap between end-to-end approaches and insertion heuristics can be brought closer by training models efficiently from trivially small TSP and transferring the learnt policy to larger graphs in a *zero-shot* fashion or via fast fine-tuning. Thus, identifying promising inductive biases,

architectures and learning paradigms that enable such zero-shot generalization to large and more complex instances is a key concern for developing practical solvers for real-world combinatorial problems.

We perform the first principled investigation into zero-shot generalization for learning large scale TSP, unifying state-of-the-art architectures and learning paradigms into one experimental pipeline. Our findings suggest that key design choices such as Graph Neural Network layers, normalization schemes, graph sparsification, and learning paradigms need to be explicitly re-designed to consider out-of-distribution generalization.

Future work can tackle generalization to large-scale problem instances in the following ways: (1) GNN architectures which are sufficiently expressive beyond simple max/mean aggregation functions, while at the same time incorporating inductive biases which account for the shifting graph degree distribution and statistics that characterize larger scale combinatorial problems. (2) Novel learning paradigms which focus on generalization, *e.g.* this work explored zero-shot generalization to larger problems, but the logical next step is to fine-tune the model on a small number of larger problems. Thus, it will be interesting to explore fine-tuning/generalization as a meta-learning problem, wherein the goal is to train model parameters specifically for fast adaptation and fine-tuning to new data distributions and problem sizes.

References

1 David Applegate, Ribert Bixby, Vasek Chvatal, and William Cook. Concorde tsp solver, 2006.

2 David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.

3 Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint*, 2016. `arXiv:1607.06450`.

4 Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *International Conference on Learning Representations*, 2017. URL: `https://openreview.net/pdf?id=Bk9mxlSFx`.

5 Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv preprint*, 2018. `arXiv:1811.06128`.

6 Xavier Bresson and Thomas Laurent. An experimental study of neural networks for variable graphs. In *International Conference on Learning Representations*, 2018.

7 Xavier Bresson and Thomas Laurent. A two-step graph convolutional decoder for molecule generation. In *NeurIPS Workshop on Machine Learning and the Physical Sciences*, 2019.

8 Quentin Cappart, Emmanuel Goutierre, David Bergman, and Louis-Martin Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1443–1451, 2019.

9 Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. In *Advances in Neural Information Processing Systems*, pages 6278–6289, 2019.

10 Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Veličković. Principal neighbourhood aggregation for graph nets. *arXiv preprint*, 2020. `arXiv:2004.05718`.

11 George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

12 Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the tsp by policy gradient. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 170–181. Springer, 2018.

13 Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *arXiv preprint*, 2020. `arXiv:2003.00982`.

14 Aaron Ferber, Bryan Wilder, Bistra Dilkina, and Milind Tambe. Mipaal: Mixed integer program as a layer. In *AAAI Conference on Artificial Intelligence*, 2020.

15 Antoine François, Quentin Cappart, and Louis-Martin Rousseau. How to evaluate machine learning approaches for combinatorial optimization: Application to the travelling salesman problem. *arXiv preprint*, 2019. arXiv:1909.13121.

16 Zhang-Hua Fu, Kai-Bin Qiu, Meng Qiu, and Hongyuan Zha. Targeted sampling of enlarged neighborhood via monte carlo tree search for {tsp}, 2020. URL: https://openreview.net/forum?id=ByxtHCVKwB.

17 Vikas K Garg, Stefanie Jegelka, and Tommi Jaakkola. Generalization and representational limits of graph neural networks. *arXiv preprint*, 2020. arXiv:2002.06157.

18 Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *arXiv preprint*, 2019. arXiv:1906.01629.

19 Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.

20 Rafael Gómez-Bombarelli, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.

21 Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020. URL: https://openreview.net/forum?id=rygGQyrFvH.

22 John J Hopfield and David W Tank. "neural" computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.

23 Jiayi Huang, Mostofa Patwary, and Gregory Diamos. Coloring big graphs with alphagozero. *arXiv preprint*, 2019. arXiv:1902.10162.

24 Gurobi Optimization Inc. Gurobi optimizer reference manual. *URL http://www. gurobi. com*, 2015.

25 Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint*, 2015. arXiv:1502.03167.

26 Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *International Conference on Machine Learning*, pages 2323–2332, 2018.

27 Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint*, 2019. arXiv:1906.01227.

28 Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

29 Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

30 Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019. URL: https://openreview.net/forum?id=ByxBFsRqYm.

31 Jan Karel Lenstra and AHG Rinnooy Kan. Some simple applications of the travelling salesman problem. *Journal of the Operational Research Society*, 26(4):717–733, 1975.

32 Ron Levie, Michael M Bronstein, and Gitta Kutyniok. Transferability of spectral graph convolutional neural networks. *arXiv preprint*, 2019. arXiv:1907.12972.

33 Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, pages 539–548, 2018.

34 John DC Little, Katta G Murty, Dura W Sweeney, and Caroline Karel. An algorithm for the traveling salesman problem. *Operations research*, 11(6):972–989, 1963.

35 Qiang Ma, Suwen Ge, Danyang He, Darshan Thaker, and Iddo Drori. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. In *AAAI Workshop on Deep Learning on Graphs: Methodologies and Applications*, 2020.

36 Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. ACM, 2019.

37 Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 2430–2439. JMLR.org, 2017.

38 Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, pages 9861–9871, 2018.

39 Alex Nowak, David Folqué, and Joan Bruna Estrach. Divide and conquer networks. In *6th International Conference on Learning Representations, ICLR 2018*, 2018.

40 Alex Nowak, Soledad Villar, Afonso S Bandeira, and Joan Bruna. A note on learning algorithms for quadratic assignment with graph neural networks. *arXiv preprint*, 2017. `arXiv:1706.07450`.

41 Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Regal: Transfer learning for fast optimization of computation graphs. *arXiv preprint*, 2019. `arXiv:1905.02494`.

42 Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint*, 2019. `arXiv:1910.10683`.

43 Maithra Raghu and Eric Schmidt. A survey of deep learning for scientific discovery. *arXiv preprint*, 2020. `arXiv:2003.11755`.

44 Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Approximation ratios of graph neural networks for combinatorial problems. In *Advances in Neural Information Processing Systems*, pages 4081–4090, 2019.

45 John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint*, 2017. `arXiv:1707.06347`.

46 Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. *arXiv preprint*, 2018. `arXiv:1802.03685`.

47 Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, pages 1–5, 2020.

48 Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

49 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

50 Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. URL: `https://openreview.net/forum?id=rJXMpikCZ`.

51 Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2020. URL: `https://openreview.net/forum?id=SkgKO0EtvS`.

52 Clement Vignac, Andreas Loukas, and Pascal Frossard. Building powerful and equivariant graph neural networks with message-passing. *arXiv preprint*, 2020. `arXiv:2006.15107`.

53 Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

54 Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint*, 2019. arXiv:1909.01315.

55 Bryan Wilder, Bistra Dilkina, and Milind Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1658–1665, 2019.

56 Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

57 Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

58 Zhihao Xing and Shikui Tu. A graph neural network assisted monte carlo tree search approach to traveling salesman problem, 2020. URL: https://openreview.net/forum?id=Syg6fxrKDB.

59 Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint*, 2018. arXiv:1810.00826.

60 Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In *International Conference on Learning Representations*, 2019.

61 Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. *arXiv preprint*, 2020. arXiv:2009.11848.

62 Emre Yolcu and Barnabas Poczos. Learning local search heuristics for boolean satisfiability. In *Advances in Neural Information Processing Systems*, pages 7990–8001, 2019.

63 Jiaxuan You, Bowen Liu, Zhitao Ying, Vijay Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. In *Advances in neural information processing systems*, pages 6410–6421, 2018.

64 Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint*, 2016. arXiv:1611.03530.

65 Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, et al. Gdp: Generalized device placement for dataflow graphs. *arXiv preprint*, 2019. arXiv:1910.01578.

A Additional Context for Figure 1: Computational challenges of learning large scale TSP

Experimental Setup. In Figure 1, we illustrate the computational challenges of learning large scale TSP by comparing three identical models trained on 12.8 Million TSP instances via reinforcement learning. Our experimental setup largely follows Section 4.1. All models use identical configurations: autoregressive decoding and Graph ConvNet encoder with Max aggregation and LayerNorm. The TSP20-50 model is trained using the greedy rollout baseline [30] and the Adam optimizer with batch size 128 and learning rate $1e-4$. Direct training, active search and finetuning on TSP200 samples is done using learning rate $1e-5$, as we found larger learning rates to be unstable. During active search and finetuning, we use an exponential moving average baseline, as recommended by Bello et al. [4].

Furthest Insertion Baseline. We characterize "good" generalization across our experiments by the well-known *furthest insertion* heuristic, which constructively builds a solution/partial tour π' by inserting node i between tour nodes $j_1, j_2 \in \pi'$ such that the distance from node i to its nearest tour node j_1 is maximized. The Appendix of Kool et al. [30] provides a detailed description of insertion heuristic approaches.

We motivate our work by showing that learning from large TSP200 is intractable on university-scale hardware, and that efficient pre-training on trivial TSP20-50 enables models to better generalize to TSP200 in a zero-shot manner. Within our computational budget, furthest insertion still outperforms our best models. At the same time, we are not claiming that it is *impossible* to outperform insertion heuristics with current approaches: reinforcement learning-driven approaches will only continue to improve performance with more computation, training data and sample efficient learning algorithms. We want to use simple non-learnt baselines to motivate the development of better architectures, learning paradigms and evaluation protocols for neural combinatorial optimization.

Routing Problems and Generalization. It is worth mentioning why we chose to study TSP in particular. Firstly, TSP has stood the test of time in terms of relevance and continues to serve as an engine of discovery for general purpose techniques in applied mathematics [11, 34, 22].

TSP and associated routing problems have also emerged as a challenging testbed for learning-driven approaches to combinatorial optimization. Whereas generalization to problem instances larger and more complex than those seen in training has at least partially been demonstrated on non-sequential problems such as SAT, MaxCut, Minimum Vertex Cover [28, 33, 46][3], the same architectures do not show strong generalization for TSP. For example, furthest insertion heuristics outperforms or are competitive with state-of-the-art approaches for TSP above tens of nodes, see Figure D.1.(e, f) from Khalil et al. [28] or Figure 5 from Kool et al. [30], despite using more computation and data than our controlled study.

B Hardware and Timings

Fairly timing research code can be difficult due to differences in libraries used, hardware configurations and programmer skill. In Table 1, we report approximate total training time and inference time across TSP sizes for the model setup described in Section 4.1. All experiments were implemented in PyTorch and run on an Intel Xeon CPU E5-2690 v4 server and four Nvidia 1080Ti GPUs. Four experiments were run on the server at any given time (each using a single GPU). Training time may vary based on server load, thus we report the lowest training time across several runs in Table 1.

We experimented with improving the latency of GNN-based models by using graph machine learning libraries such as DGL [54]. DGL requires graphs to be prepared as sparse library-specific data objects, which significantly boosts the inference speed of GNNs. However, using DGL had a negative impact on the speed of the rest of our pipeline (batched data preparation, decoders, beam search). This issue is especially amplified for reinforcement learning, where we constantly generate new random datasets at each epoch. For now, we present timings and results with pure PyTorch code. We confirm that results are consistent with using DGL, but decided against it in order to run a large volume of experiments for more comprehensive analysis.

C Datasets

We generate 2D Euclidean TSP instances of varying sizes and complexities as graphs of n node locations sampled uniformly in the unit square $S = \{x_i\}_{i=1}^{n}$ and $x_i \in [0, 1]^2$. For supervised learning, we generate a training set of 1,280,000 samples each for TSP20, TSP50,

[3] It is worth noting that classical algorithmic and symbolic components such as graph reduction, sophisticated tree search as well as post-hoc local search have been pivotal and complementary to GNNs in enabling such generalization [33].

Table 1 Approximate training time (12.8M samples) and inference time (1,280 samples) across TSP sizes and search settings for SL and RL-trained models. *GS*: Greedy search, *BS128*: beam search with width 128, *S128*: sampling 128 solutions. RL training uses the rollout baseline and timing includes the time taken to update the baseline after each 128,000 samples.

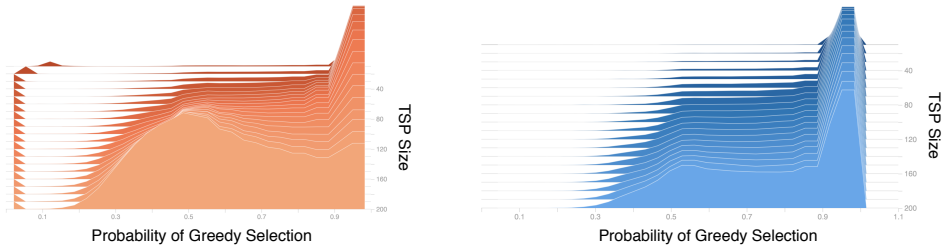
Graph Size	Training Time		Inference Time		
	SL	RL	GS	BS128	S128
TSP20	4h 24m	8h 02m	2.62s	7.06s	63.37s
TSP20-50	9h 49m	15h 47m	-	-	-
TSP50	16h 11m	40h 29m	7.45s	29.09s	86.48s
TSP100	68h 34m	108h 30m	19.04s	98.26s	180.30s
TSP200	-	495h 55m	54.88s	372.09s	479.37s

TSP100, and TSP20-50. The groundtruth tours are obtained using the Concorde solver [1]. For reinforcement learning, 128,000 samples are randomly generated for each epoch (without optimal solutions). We compare models on a held-out test set of 25,600 TSP samples and their corresponding optimal tours, consisting of 1,280 instances each of TSP10, TSP20, ..., TSP200. We release all dataset files as well as the associated scripts to produce TSP datasets of arbitrarily large sizes along with our open-source codebase.

D Learning Paradigms and Amenity to Search

Figure 10 demonstrate that SL models are more amenable to beam search and sampling, but are outperformed by RL-rollout models under greedy search. In Figure 12, we investigate the impact of learning paradigms on probability distributions by plotting histograms of the probabilities of greedy selections during inference across TSP sizes for identical models trained with SL and RL. We find that the rollout baseline, which encourages better greedy behaviour, leads to the model making very confident predictions about selecting the next node at each decoding step, even beyond training size range. In contrast, SL models are trained with teacher forcing, *i.e.* imitating the optimal solver at each step instead of using their own prediction. This results in less confident predictions and poor greedy decoding, but makes the probability distribution more amenable to beam search and sampling techniques.

We understand this phenomenon as follows: More confident predictions (Figure 12b) do not automatically imply better solutions. However, sampling repeatedly or maintaining the top-b most probable solutions from such distributions is likely to contain very similar tours. On the other hand, less sharp distributions (Figure 12a) are likely to yield more diverse tours with increasing b. This may result in comparatively better optimality gap, especially for TSP sizes larger than those seen in training.



(a) Supervised Learning **(b)** Reinforcement Learning

Figure 12 Histograms of greedy selection probabilities (x-axis) across TSP sizes (y-axis).

E Visualizing Model Predictions

As a final note, we present a visualization tool for generating model predictions and heatmaps of TSP instances, see Figures 13 and 14. We advocate for the development of more principled approaches to neural combinatorial optimization, *e.g.* along with model predictions, visualizing the reduce costs for each edge (obtained using the Gurobi solver [24]) may help debug and improve learning-driven approaches in the future.

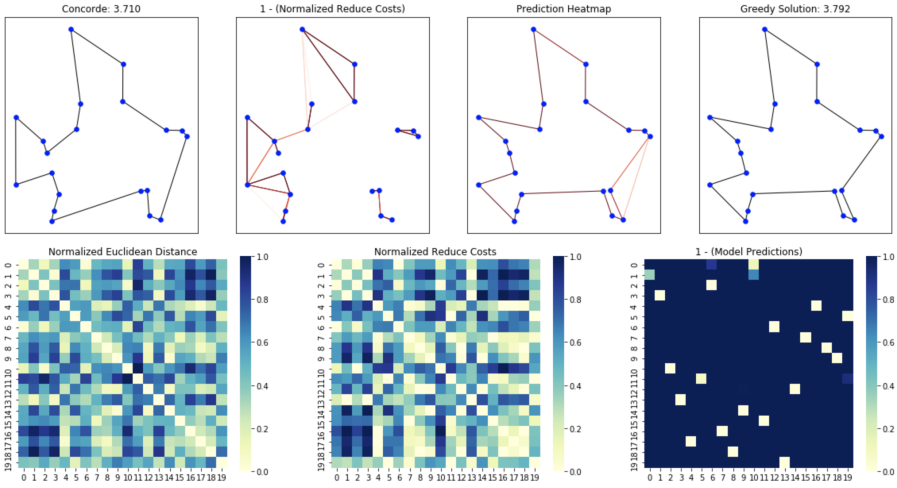


Figure 13 Prediction visualization for TSP20.

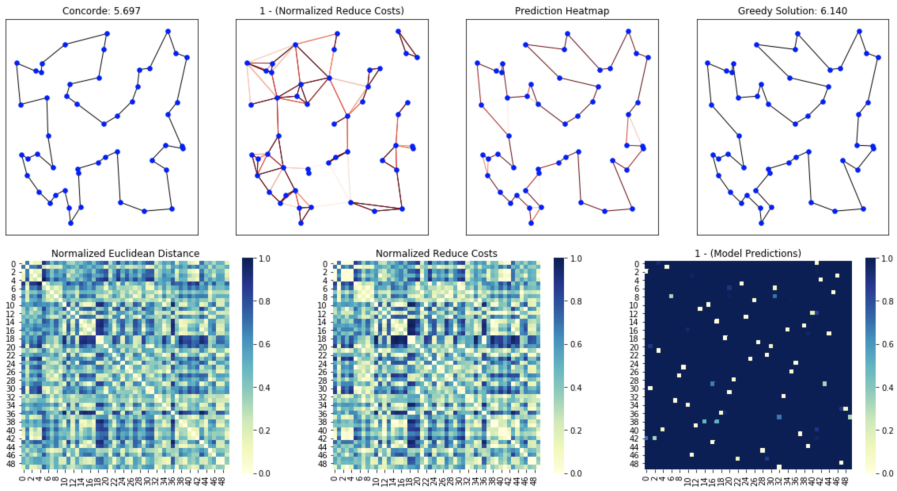


Figure 14 Prediction visualization for TSP50.

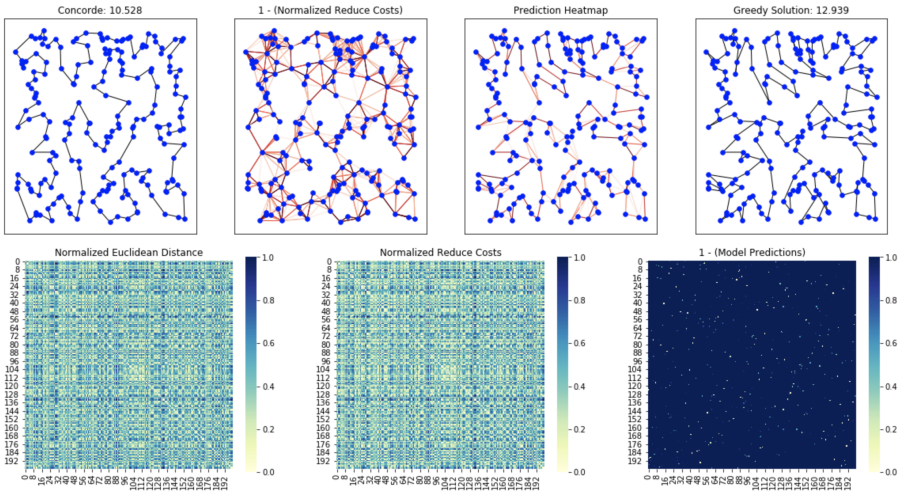


Figure 15 Prediction visualization for TSP200.