GRAPH NEURAL NETWORK GUIDED LOCAL SEARCH FOR THE TRAVELING SALESPERSON PROBLEM

Anonymous authors
Paper under double-blind review

ABSTRACT

Solutions to the Traveling Salesperson Problem (TSP) have practical applications to processes in transportation, logistics, and automation, yet must be computed with minimal delay to satisfy the real-time nature of the underlying tasks. However, solving large TSP instances quickly without sacrificing solution quality remains challenging for current approximate algorithms. To close this gap, we present a hybrid data-driven approach for solving the TSP based on Graph Neural Networks (GNNs) and Guided Local Search (GLS). Our model predicts the regret of including each edge of the problem graph in the solution; GLS uses these predictions in conjunction with the original problem graph to find solutions. Our experiments demonstrate that this approach converges to optimal solutions at a faster rate than state-of-the-art learning-based approaches and non-learning GLS algorithms for the TSP, notably finding optimal solutions to 96% of the 50-node problem set, 7% more than the next best benchmark, and to 20% of the 100-node problem set, 4.5× more than the next best benchmark. When generalizing from 20-node problems to the 100-node problem set, our approach finds solutions with an average optimality gap of 2.5%, a 10× improvement over the next best learning-based benchmark.

1 INTRODUCTION

Sixty years ago, the route of a delivery truck would have been fixed well before the truck departed the warehouse. Today, thanks to the availability of real-time traffic information, and the option to transmit instructions to the driver to add or remove delivery locations on-the-fly, the route is no longer fixed. Nevertheless, minimizing the length or duration of the route remains an important problem. This is an instance of the Traveling Salesperson Problem (TSP), and one of a growing number of practical applications which require solving combinatorial optimization problems in real time. In these problems, there is often a cost attributed to waiting for an optimal solution or hard deadlines at which decisions must be taken. For example, the driver cannot wait for a new solution to be computed – they may miss their deliveries, or the traffic conditions may change again. Therefore, there is a pressing need for combinatorial optimization algorithms which produce *high-quality solutions with minimal computation time.* This remains challenging for current algorithms, as most research has focused on finding optimal solutions to static problems with little regard for computation time.

Contributions We present a hybrid data-driven approach for approximately solving the Euclidean TSP based on Graph Neural Networks (GNNs) and Guided Local Search (GLS), which we demonstrate converges to optimal solutions more quickly than three recent learning-based approaches and one non-learning GLS algorithm. We provide the following contributions:

- Our GLS algorithm is guided by the *global regret* of each edge in the problem graph. This allows the algorithm to differentiate edges that are very costly to include in the solution from ones that are not so costly, whether or not they are part of the optimal solution. Thus, using regret allows us to find high-quality, rather than optimal, solutions very quickly. We are the first to use a measure of global regret, which we define as the cost of enforcing a decision relative to the cost of an optimal solution.
- We make this computationally tractable by approximating regret with a learned representation. Our GNN-based model operates on the *line graph* of the problem graph, which allows us to build a network without node features, focusing only on edge weight.

- We present an improved evaluation criterion for the TSP that emphasizes the trade-off between solution quality and computation time. Notably, our approach finds optimal solutions to 96% of the 50-node problem set, 7% more than the next best benchmark, and to 20% of the 100-node problem set, 4.5× more than the next best benchmark. When generalizing from 20-node problems to the 100-node problem set, our approach finds solutions with an average optimality gap of 2.5%, a 10× improvement over the next best learning-based benchmark.

2 Related Work

The Operations Research (OR) community is responsible for the majority of research in algorithms for solving combinatorial optimization problems. Concorde (Applegate et al., 2006) is widely regarded as the best exact TSP solver. As an exact solver, it can guarantee the level of optimality of the solutions it finds. It uses cutting plane algorithms (Dantzig et al., 1954; Padberg & Rinaldi, 1990; Applegate et al., 2003) and branch-and-cut methods to iteratively prune away parts of search space that will not contain an optimal solution. LKH-3 (Helsgaun, 2017), and its predecessor, LKH-2 (Helsgaun, 2009), are approximate TSP and Capacitated Vehicle Routing Problem (CVRP) solvers based on the κ-opt heuristic (Lin & Kernighan, 1973). While these solvers do not provide any guarantees, experimentation has demonstrated that they are extremely effective. Nevertheless, these methods focus on finding optimal solutions with little regard for computation time, making them ill-adapted for solving the TSP in real-time. With this in mind, Arnold & Sörensen (2019a) introduces KGLS, a GLS algorithm for the CVRP that is guided by three engineered features. They demonstrate that their algorithm can find solutions almost as good as those found by state-of-the-art heuristics in a fraction of the time.

As noted by Bengio et al. (2021); Cappart et al. (2021), classical combinatorial optimization approaches solve each problem instance in isolation, overlooking the fact that in practice, problems and their solutions stem from related data distributions. Machine learning offers a way to exploit this observation. Neural network-based methods for solving combinatorial optimization problems have existed for more than three decades (Smith, 1999), yet new neural architectures, especially GNNs, have driven a surge of activity in this area. We classify learning-based approaches for solving the TSP into the three categories identified by Bengio et al. (2021); our approach belongs to the second category, described below.

ML alone provides a solution. These approaches use a machine learning model to output solutions directly from the problem definition. Vinyals et al. (2015) propose PtrNet, a sequence-to-sequence model based on LSTMs, which iteratively constructs a solution by outputting the permutation of the input nodes. They train their model using expert solutions from Concorde. They use beam search (Reddy, 1977) to produce better solutions than those sampled from their model directly. Bello et al. (2016) present a method of training PtrNet without supervision using policy gradients. Their "active search" method resembles early neural network-based approaches to combinatorial optimization, in which a model was trained online to solve a particular problem instance (Smith, 1999). This method effectively solves each problem in isolation, thus it does not leverage expertise extracted from distribution of training problem instances. Kool et al. (2018) take a similar approach to Bello et al. (2016) but use a Transformer architecture (Vaswani et al., 2017) rather than an LSTM. Their model can also be seen as a Graph Attention Network (GAT) (Veličković et al., 2017) applied to a complete graph. Joshi et al. (2019) train a model based on Graph Convolutional Networks (GCNs) (Kipf & Welling, 2016) to predict which edges in the TSP graph are likely to be part of the optimal solution, and reconstruct valid tours using beam search. Similar to Vinyals et al. (2015), they train their model using expert solutions from Concorde.

ML provides information to an OR algorithm. Machine learning may not be suitable to solve a combinatorial optimization problem alone. Instead, machine learning can provide information to augment a combinatorial optimization algorithm. Deudon et al. (2018) use a model based on the Transformer architecture to construct solutions iteratively and train it using policy gradients (they take a similar approach to Kool et al., 2018). They apply a 2-opt local search heuristic to solutions sampled from their model. In contrast to prior work, we are the first to use a machine learning model to approximate *regret*. Furthermore, previous work in this category has only leveraged heuristics, whereas we use a model to provide information to a metaheuristic (GLS).

ML makes decisions within an OR algorithm. Finally, a machine learning model can be embedded inside a combinatorial optimisation algorithm. In this paradigm, a master algorithm controls the high-level procedure while repeatedly calling a machine learning model to make lower level decisions. Dai et al. (2017) present a model that uses a GNN to learn an embedding of the current partial solution and a DQN (Mnih et al., 2015) to iteratively select nodes to insert using a cheapest insertion heuristic. They also use an "active search" method when solving the TSP. Wu et al. (2020) and da Costa et al. (2020) present policy gradient algorithms to learn policies to apply 2-opt operations to existing feasible solutions. Wu et al. (2020) use a Transformer architecture. da Costa et al. (2020) use a combination of GCN and RNN modules, and acheive better results. These approaches can either be seen as end-to-end learning approaches belonging to the first category, or as local search procedures where an ML model decides where to apply an operator.

3 PRELIMINARIES

Traveling Salesperson Problem We focus on the Euclidean TSP, although our approach can be applied to other routing problems, such as the CVRP. A problem with n cities, typically denoted TSPn, is represented as a complete, undirected, weighted graph $G = (V, E)$ with n nodes. The edges E represent connections between cities and are weighted by the Euclidean distance between the adjacent nodes. A solution, or *tour*, is a Hamiltonian cycle: a closed path through the graph that visits every node exactly once. An optimal solution is a cycle of minimum weight.

Regret Regret measures the future cost of an action that also yields an immediate reward, and is typically used to make greedy combinatorial optimization algorithms less myopic. Typically, regret is computed over a fixed horizon. For example, Potvin & Rousseau (1993) evaluate the cost of inserting a node in the best versus second-best position when constructing a CVRP solution. Hassin & Keinan (2008) solve the TSP using regret, allowing a greedy construction heuristic to remove the most recently inserted node. It is not computationally feasible to calculate regret over a global horizon, for example, for all possible insertion sequences. However, if it were possible, a greedy algorithm could compute an optimal solution by selecting the lowest regret decision at each step.

Local Search Local Search (LS) is a general improvement heuristic. Starting from an initial solution, local search iteratively moves to *neighboring* solutions that are lower cost than the current solution according to the objective function $g(s)$. Neighboring solutions are solutions that are reachable from a given solution by applying a certain function, or *operator*. The set of all solutions reachable from another solution by applying an operator define the neighborhood of that operator. The algorithm terminates when all neighboring solutions are inferior to the current solution, meaning the search has reached a local optimum.

Guided Local Search Guided Local Search (GLS) is a metaheuristic that sits on top of LS and allows it to escape local optima (Voudouris & Tsang, 1996). To apply GLS, the designer must define some *aspects* of a solution. When trapped in a local optimum, the algorithm penalizes certain aspects of the current solution which are considered to be unfavorable. The underlying LS procedure searches using an objective function that is augmented by these penalties, thus it is incentivized to remove heavily penalized aspects from the solution. The augmented objective function $h(s)$ is

$$h(s) = g(s) + \lambda \sum_{i=0}^{M} p_i I_i(s), \tag{1}$$

where s is a solution, $g(s)$ is the objective function, λ is a scaling parameter, i indexes the aspects, M is the number of aspects, p_i is the current number of penalties assigned to aspect i, and I_i is an indication of whether s exhibits aspect i, i.e.

$$I_i(s) = \begin{cases} 1 & \text{if } s \text{ exhibits aspect } i, \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

For the TSP, aspects of the solution are often defined as the edges in the problem graph. Therefore, $I_i(s)$ indicates if an edge is in the solution s. Upon reaching a local optimum, which aspects are penalized is determined by a utility function. The utility of penalising aspect i, util$_i$, is defined as

$$\text{util}_i(s_*) = I_i(s_*) \frac{c_i}{1 + p_i}, \tag{3}$$

where s_* is the solution at a local optimum, $I_i(s_*)$ indicates if the solution exhibits aspect i, and c_i is the cost of the aspect i. The cost of an aspect measures how unfavorable it is. The higher the cost, the greater the utility of penalizing that aspect. In the context of the TSP, the cost can be the weight of the edge (Voudouris & Tsang, 1999), or a combination of various features (Arnold & Sörensen, 2019a). Conversely, the more penalties assigned to that aspect, the lower the utility of penalising it again. The aspects with the maximum utility are always penalized, which means increasing p_i by one. This penalization mechanism distributes the search effort in the search space, favoring areas where a promise is shown (Voudouris & Tsang, 1996).

We use a variation of the classical GLS algorithm (see Voudouris & Tsang, 1999) that applies alternating optimisation and perturbation phases (see Arnold & Sörensen, 2019a). During an optimization phase, the local search procedure is guided by the original objective function g. During a perturbation phase, it is guided by the augmented objection function h. After an edge is penalized, the local search is applied *only* on the penalized edge. That is, only operations that would remove the penalized edge are considered. This differs from the local search during the optimization phase, which considers all solutions in the neighborhood of the given operator. The perturbation phase continues until K operations (operations that improve the solution according to the augmented objective function h) have been applied to the current solution. These operations perturb the solution enough for the local search to escape a local minimum. The alternating phases continue until the stopping condition is met.

4 METHOD

Our hybrid method, shown in Figure 1, combines a machine learning model and a metaheuristic. Our GNN-based model learns an approximation of the *global regret* of including each edge of the problem graph in the solution. The metaheuristic, GLS, uses this learned regret conjunction with the original problem graph to quickly find high-quality solutions. The learned regret allows the algorithm to differentiate between edges which are costly to include in the solution and ones that are not so costly, thus improving its ability to steer the underlying local search procedure out of local minima and towards promising areas of the solution space.

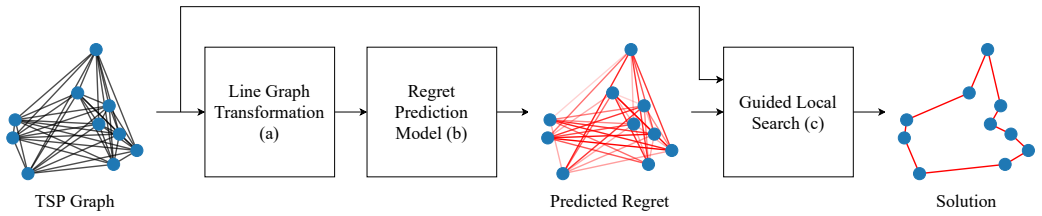


Figure 1: From a TSP formulated as a graph, we find the line graph (a) and input it into our regret approximation model (b), which predicts the regret of including each edge in the solution. GLS (c) uses these predictions in conjunction with the original problem graph to quickly find a high-quality solution.

4.1 GLOBAL REGRET

We define *global regret* as the cost of requiring a certain decision to be part of the solution relative to the cost of a globally optimal solution. Unlike previous heuristics using regret, which calculate the cost of a decision relative to some fixed number of alternatives (for example, the next best, or two next best options), our regret is measured relative to a *global optimal solution*. Decisions that are part of an optimal solution have zero regret, and all other decisions have positive regret. Mathematically,

$$r_i = \frac{g(s_i^*)}{g(s^*)} - 1, \tag{4}$$

where r_i is the regret of decision i, g is the objective function, s_i^* is an optimal solution with i fixed, and s^* a globally optimal solution. With perfect information, a greedy algorithm could construct an optimal solution by sequentially selecting the lowest-regret decisions.

(a) G (b) $L(G)$

Figure 2: An example of a graph and the corresponding line graph. The edges in G are the nodes in $L(G)$, and vice-versa.

In the TSP, decisions correspond to which edges are included in the solution, thus regret is defined as the cost of requiring that a certain edge be part of the solution. We posit that using regret is preferable to directly classifying which edges are part of the optimal solution (which is the approach taken by Joshi et al., 2019). Where classification can produce a probability that the edge is part of *optimal* solution, regret can differentiate between edges that are very costly to have as part of the solution and ones that are not so costly, whether or not they are part of the optimal solution. Thus, using regret furthers our goal of finding high-quality, rather than optimal, solutions, with minimal computation time.

4.2 Regret Approximation Model

Calculating the global regret of an edge in the TSP graph requires solving the TSP itself, which is computationally intractable. Instead, we aim to learn a function $\hat{r}_{ij}$ that approximates the regret of an edge r_{ij}. We use GNNs to achieve this, as they are universal function approximators that operate on graph-structured data.

Input transformation Typically, GNNs aggregate messages and store states on the nodes of a graph (Gilmer et al., 2017). Therefore, most approaches to edge-property prediction predict the properties of an edge as a function of the surrounding node states. Instead, we input the line graph of the original problem graph into our model. The line graph $L(G)$ of an undirected graph G is a graph such that there exists a node in $L(G)$ for every edge in G, and that for every two edges in G that share a node, there exists an edge between their corresponding nodes $L(G)$. Figure 2 illustrates this transformation for a simple graph. The result is that our model aggregates messages and stores states on the edges of the problem graph (the nodes of the line graph). This allows us to build models with no node features, which is advantageous as the edge weights, not the specific node locations, are relevant when solving the TSP. For a complete, undirected graph G with n nodes, there are $n(n-1)/2$ nodes and $n(n-1)(n-2)$ edges in $L(G)$. Thus, although G is complete, $L(G)$ can be very sparse.

Model architecture Our model consists of an embedding layer, several GNN layers, and an output layer. The embedding layer is an edge-wise fully connected layer that computes d_h-dimensional edge embeddings from d_x-dimensional edge features. Node features, if used, are concatenated onto the feature set of the adjacent edges. The forward pass of the embedding layer is written as

$$\mathbf{h}_{ij}^0 = \mathbf{W}\mathbf{x}_{ij} + \mathbf{b}, \tag{5}$$

where $\mathbf{h}_{ij}^0$ is the initial embedding of edge ij, $\mathbf{W}$ is a learnable weight matrix, $\mathbf{x}_{ij}$ are the input features of edge ij (including any node features), and $\mathbf{b}$ is a set of learnable biases. We use $d_x = 1$ and $d_h = 128$. The edge embeddings are updated using T message passing layers. Inspired by the encoder layer of Kool et al. (2018), each layer consists of multiple sublayers. The forward pass is given by

$$\dot{\mathbf{h}}_{ij}^{t+1} = f_{\text{BN}}^t(\mathbf{h}_{ij}^t + f_{\text{MHA}}^t(\mathbf{h}_{ij}^t, L(G))), \tag{6}$$

$$\mathbf{h}_{ij}^{t+1} = f_{\text{BN}}^t(\dot{\mathbf{h}}_{ij}^{t+1} + f_{\text{FF}}^t(\dot{\mathbf{h}}_{ij}^{t+1})), \tag{7}$$

where f_{MHA} is a multi-headed GAT layer (Veličković et al., 2017), f_{FF} is a feedforward layer, f_{BN} is batch normalisation (Ioffe & Szegedy, 2015), and $\dot{\mathbf{h}}_u^{t+1}$ is a hidden state. The layers do not share parameters. The GAT layer uses $M = 8$ heads and dimensionality $d_h/M = 16$, and the FF layer uses one hidden sublayer with dimensionality 512 and ReLU activation. Finally, the output

layer is a single edge-wise fully connected layer that computes a one-dimensional output from the d_h-dimensional node embeddings computed by the final message passing layer. This is written as

$$\hat{r}_{ij} = \mathbf{W}\mathbf{h}_{ij}^T + \mathbf{b}, \tag{8}$$

where $\hat{r}_{ij}$ is the output for edge ij and $\mathbf{h}_{ij}^T$ is the final embedding of that edge.

4.3 Regret-Guided Local Search

We adapt GLS to use regret to solve the TSP, including how the initial solution is built, the local search procedure, and the perturbation strategy. Our GLS uses alternating optimization and perturbation phases. During an optimization phase, the local search procedure greedily accepts changes to the solution until it reaches a local minimum. During a perturbation phase, the algorithm penalizes and attempts to remove edges in the current solution with high regret, thus allowing it to escape local minima while simultaneously guiding it towards promising areas of the solution space, i.e., those with low regret. Effectively, the regret predictions produced by our model allow our GLS algorithm to undo costly decisions made during the greedy optimization phase.

Initial solution We use a greedy nearest neighbor algorithm to construct an initial solution. Beginning from the origin node we iteratively select the lowest-regret edge leading to an unvisited node, until all nodes have been visited.

Local Search neighborhoods Our LS procedure uses two solution operators for the TSP, *relocate* and *2-opt*. It alternates between using either operator, and uses a "best improvement" strategy, meaning that it exhaustively searches the neighborhood corresponding with the current operator and accepts the solution that improves the objective function the most, before continuing with the other operator. The algorithm terminates when no improvement can be found in either neighborhood. The relocate operator simply changes the position of a single node in the tour. The 2-opt operator selects two nodes in the tour to swap. This divides the tour into three segments: an initial segment, an intermediate segment, and a final segment. The tour is reassembled beginning with the initial segment, the intermediate segment in reverse order, and the final segment. It is a special case of the κ-opt operator (Lin & Kernighan, 1973), although it was introduced earlier (Croes, 1958).

Perturbation strategy We define the cost of an edge as the predicted regret $\hat{r}_{ij}$ of that edge. The utility of penalizing edge ij, util_{ij}, is therefore defined as

$$\text{util}_{ij}(s_*) = I_{ij}(s_*)\frac{\hat{r}_{ij}}{1 + p_{ij}}, \tag{9}$$

where we remind the reader that s_* is the solution at a local optimum, $I_{ij}(s_*)$ indicates if the solution contains edge ij, and p_{ij} is the number of penalties assigned to that edge. The edges of maximum utility are penalized. Afterward, the local search is applied *only* on the penalized edge. That is, only operations that would remove the penalized edge are considered.

5 Experiments

We train our model using supervised learning. It uses a single input feature, edge weight (i.e. the Euclidean distance between adjacent nodes), to predict the regret of the edges in the problem graph. Further details of our implementation are found in Appendix A. While further input features could be considered, they come at a computational cost, as seen in Table 3. In Appendix C we compare 11 input features, and demonstrate that edge weight is the most important feature when predicting the regret of an edge.

Evaluation We evaluate the trade-off between solution quality and computation time when solving a set of TSP instances by fixing computation time and measuring the solution quality, in terms of the median optimality gap and the number of problems solved optimally. *Previous works leave both variables unfixed*, meaning that approaches cannot be compared to one another.

We compare our approach to the following:

- Local Search (Baseline)
- *Attention, learn to solve routing problems!* (Kool et al., 2018)

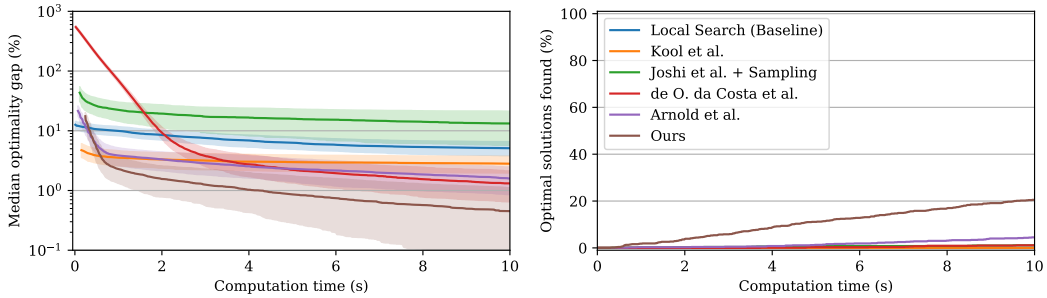


Figure 3: Solution quality as a function of time when solving the 100-node problem set, for our approach and benchmarks, demonstrating that our method converges to optimal solutions at a faster rate than the evaluated benchmarks. The left plot shows the median optimality gap (trace) and inter-quartile range (fill). The right plot shows the percentage of optimally solved problems.

- *An efficient graph convolutional network technique for the travelling salesman problem* (Joshi et al., 2019)
- *Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning* (da Costa et al., 2020)
- *Knowledge-guided local search for the vehicle routing problem* (Arnold & Sörensen, 2019a)

Where available, we use publicly available implementations and pre-trained models. We modified the implementations of Kool et al. (2018); Joshi et al. (2019); da Costa et al. (2020) to stop when the computation time limit is reached, rather than after a fixed number of iterations. We also modified these implementations to report more precise tour lengths, as these are necessary to accurately report the number of problems solved optimally. Joshi et al. (2019) uses beam search, which is not an anytime algorithm. Therefore, we modified their implementation to use an anytime sampling approach similar to Kool et al. (2018). There is no publicly available implementation of Arnold & Sörensen (2019a), so we use the GLS implementation used in our approach with the guides described in their paper. Finally, we compare the local search procedure to a baseline.

We evaluate three problem sets, TSP20, TSP50, and TSP100, consisting of one thousand 20, 50, and 100-node 2D Euclidean TSP instances, respectively. We generate the problem instances by uniform-randomly sampling node locations in the unit square $[0, 1]^2$, which is inline with the methods used by Kool et al. (2018); Joshi et al. (2019); da Costa et al. (2020). We allow up to 10 seconds to solve each problem, including the time required to calculate input features, evaluate a model, or to construct an initial solution. We evaluate the various approaches one-at-a-time, to ensure they do not compete for computational resources. While the evaluated approaches can all be parallelized, either on the CPU, on the GPU, or both, not all of the publicly available implementations are parallelized. We aim to evaluate the performance of the algorithms, not their implementations, therefore we allow each approach to solving the problem instances one-at-a-time (i.e., a batch size of $B = 1$). Based on approximately two hundred thousand TSP solutions, we determined that an absolute threshold of 10^{-7} on the value of the objective function is sufficient to separate optimal and sub-optimal solutions.

Results The performance of each approach at the computation time limit is shown in Table 1. Figure 3 shows the performance as a function of time when solving the 100-node problem set. Similar plots depicting the performance of a function of time for all problem sets are found in Appendix B. All approaches achieve reasonably good results on the 20-node problem set, however performance decreases as problem size increases. Our approach outperforms the others on all problem sets. Notably, our approach finds optimal solutions to 96% of the TSP50 problem set, 7% more than the next best benchmark (Arnold & Sörensen, 2019a), which finds 90%. When solving the TSP100 problem set, our approach finds optimal solutions to 20.5% of the problems, 4.5× more than the next best benchmark (Arnold & Sörensen, 2019a) and achieves an average optimality gap of 0.45% when solving the TSP100 problem set, 2.9× better than the next best benchmark (also Arnold & Sörensen, 2019a), which achieves an average optimality gap of 1.59%.

Problem	Method	Optimality gap (%) Median	IQR	Optimal solutions (%)
TSP20	Local Search (Baseline)	0.0000	0.6951	42.6426
	Kool et al.	0.0000	0.0000	81.2813
	Joshi et al. + Sampling	0.0000	0.0000	98.1982
	de O. da Costa et al.	0.0000	0.0000	99.2993
	Arnold et al.	**0.0000**	**0.0000**	**100.0000**
	Ours	**0.0000**	**0.0000**	**100.0000**
TSP50	Local Search (Baseline)	2.5284	2.9722	7.4074
	Kool et al.	0.4279	0.9597	18.9189
	Joshi et al. + Sampling	0.0000	0.1278	67.0671
	de O. da Costa et al.	0.0297	0.3502	46.6466
	Arnold et al.	0.0000	0.0000	90.1902
	Ours	**0.0000**	**0.0000**	**96.1962**
TSP100	Local Search (Baseline)	5.1069	2.7396	0.0000
	Kool et al.	2.8101	1.7887	0.0000
	Joshi et al. + Sampling	13.2537	15.5596	1.1011
	de O. da Costa et al.	1.3148	1.4757	1.1011
	Arnold et al.	1.5911	1.6178	4.5045
	Ours	**0.4513**	**1.0585**	**20.5205**

Table 1: Evaluation of our method and benchmarks on 20, 50, and 100-node TSP problem sets after 10 seconds of computation time per instance. The median and inter-quartile range of the gap between the best and optimal solutions, and the percentage of optimally solved problems are reported.

Model	Problem	Method	Optimality gap (%) Median	IQR	Optimal solutions (%)
TSP20	TSP50	Kool et al.	2.5270	2.1080	0.5005
		Joshi et al. + Sampling	28.2036	5.7565	0.0000
		de O. da Costa et al.	3.2976	2.7561	0.5005
		Ours	**0.0000**	**0.0618**	**73.1732**
TSP20	TSP100	Kool et al.	26.5113	4.6053	0.0000
		Joshi et al. + Sampling	89.1632	7.4764	0.0000
		de O. da Costa et al.	32.9608	12.1003	0.0000
		Ours	**2.5129**	**2.1176**	**1.2012**
TSP50	TSP100	Kool et al.	4.0025	1.6081	0.0000
		Joshi et al. + Sampling	56.7009	6.6427	0.0000
		de O. da Costa et al.	2.2806	1.8278	0.0000
		Ours	**1.1577**	**1.4869**	**7.6076**

Table 2: Evaluation of our method and learning-based benchmarks on 50, and 100-node TSP problem sets using models trained on 20, and 50-node problems after 10 seconds of computation time per instance. The median and inter-quartile range of the gap between the best and optimal solutions, and the percentage of optimally solved problems are reported.

We evaluate the performance of our approach and other learning-based approaches when generalizing from smaller problem instances to larger ones. The performance of each approach at the computation time limit is reported in Table 2.

All approaches perform worse when using models trained on smaller problems. Our approach performs the best, notably finding optimal solutions to 73% of the problems in the TSP50 problem set when solving them using a model trained on 20-node problems, when the next best approaches find 0.5% (Kool et al., 2018; da Costa et al., 2020). Our model trained on 20-node problems achieves an average optimality gap of 2.5% when solving the TSP100 problem set, a 10$\times$ improvement over the next best benchmark (Kool et al., 2018), which achieves an average optimality gap of 26.5%.

6 DISCUSSION

Our experiments demonstrate that our approach converges to optimal solutions at a faster rate than three recent learning-based approaches (Kool et al., 2018; Joshi et al., 2019; da Costa et al., 2020) and one recent non-learning GLS algorithm (Arnold & Sörensen, 2019a) when solving the Euclidean TSP. The results presented in Table 1 and Table 2 align with the results presented by Joshi et al. (2019) for their own method and the method of Kool et al. (2018), despite the difference in evaluation criteria. Using our evaluation criterion we achieve slightly worse results in generalization than those reported by Kool et al. (2018) in their paper, which may be attributed to evaluating the model for a fixed computation time, which we do here, rather than a fixed number of iterations: models evaluated on larger problems are slower to run, thus find fewer solutions in a fixed computation time.

Our approach generalizes to larger problems better than the other learning-based approaches, which may be as a result of the unique input transformation we apply to the problem graph. We input the line graph $L(G)$ of the problem graph G, which allows the model to aggregate messages and store states on edges rather than nodes. This allows us to build models without node features, which is advantageous as the edge weights, not the specific node positions, are relevant when solving the TSP. Including the node positions as features, as done by Kool et al. (2018); Joshi et al. (2019); da Costa et al. (2020) may hinder the learned policy's ability to generalize. Deudon et al. (2018) apply PCA on the node positions before inputting them into their model so they are rotation invariant, yet they do not report generalization performance. For a Euclidean TSP problem graph with n nodes, $L(G)$ has $n(n-1)/2$ nodes, meaning that although G is complete, $L(G)$ can be very sparse, which may help learning (Cappart et al., 2021). However, the model consumes more GPU memory than models using the problem graph G.

Many approaches that learn construction heuristics (for example Kool et al., 2018) treat the tour as a permutation of the input nodes. By considering the solution as a sequence, these approaches miss out on its symmetry: a TSP solution is invariant to the origin city. Outputting the correct sequence, even when augmented by beam search, becomes increasingly difficult as the number of cities increases. Our model learns the regret of including a given edge in the solution based on its features and those of its neighbors. This learning task implicitly assumes the symmetry of a TSP solution and leverages the strengths of GNNs.

Our approach is not specific to the TSP, however one disadvantage is that it relies on an explicitly designed local search algorithm. To apply our framework to another problem, the local search procedure must be modified with operators that are appropriate for that problem. For example, a local search procedure for the CVRP requires intra-route operators (such as 2-opt and relocate) and inter-route operators. In future work, our regret approximation model could be combined with a general learned local search algorithm and could be trained end-to-end.

7 CONCLUSION

We present a hybrid data-driven approach for solving the TSP based on Graph Neural Networks (GNNs) and Guided Local Search (GLS). To inform GLS, we use a GNN-based model to compute fast, generalizable approximations of the *global regret* of each edge. Our model operates on the line graph of the TSP graph, which allows us to build a network that uses edge weight alone as an input feature. This approach allows our algorithm to escape local minima and find high-quality solutions with minimal computation time. Finally, we present an improved evaluation criterion for algorithms solving combinatorial optimization problems that emphasizes the trade-off between solution quality and computation time. Using this criterion, we demonstrate that our approach converges to optimal solutions at a faster rate than three learning-based approaches and one non-learning GLS algorithm for the TSP.

REFERENCES

David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems. *Mathematical Programming*, 97(1):91–153, 2003.

David Applegate, Robert Bixby, Vašek Chvatál, and William Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, Princeton, NJ, 2006.

Florian Arnold and Kenneth Sörensen. Knowledge-guided local search for the vehicle routing problem. *Computers & Operations Research*, 105:32–46, 2019a.

Florian Arnold and Kenneth Sörensen. What makes a VRP solution good? The generation of problem-specific knowledge for heuristics. *Computers & Operations Research*, 106:280–288, 2019b.

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2): 405–421, 2021.

Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint arXiv:2102.09544*, 2021.

G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

Paulo R de O da Costa, Jason Rhuggenaath, Yingqian Zhang, and Alp Akcay. Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. *arXiv preprint arXiv:2004.01608*, 2020.

Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *International Conference on Neural Information Processing Systems*, pp. 6351–6361, 2017.

G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954.

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

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pp. 1263–1272. PMLR, 2017.

Loïc Le Gratiet, Stefano Marelli, and Bruno Sudret. *Metamodel-Based Sensitivity Analysis: Polynomial Chaos Expansions and Gaussian Processes*, pp. 1–37. Springer International Publishing, Cham, 2016.

Refael Hassin and Ariel Keinan. Greedy heuristics with regret, with application to the cheapest insertion algorithm for the TSP. *Operations Research Letters*, 36(2):243–246, 2008.

Keld Helsgaun. General k-opt submoves for the Lin–Kernighan TSP heuristic. *Mathematical Programming Computation*, 1(2):119–163, 2009.

Keld Helsgaun. An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems. Technical report, Roskilde University, 2017.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pp. 448–456. PMLR, 2015.

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

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973.

M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2): 239–245, 1979.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, (508):529–533, 2015.

Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *Society for Industrial and Applied Mathematics*, 33(1): 60–100, 1990.

Andrei Paleyes, Mark Pullin, Maren Mahsereci, Neil Lawrence, and Javier Gonzalez. Emulation of physical processes with emukit. In *Second Workshop on Machine Learning and the Physical Sciences*. NeurIPS, 2019.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pp. 8026–8037, 2019.

Jean-Yves Potvin and Jean-Marc Rousseau. A parallel route building algorithm for the vehicle routing and scheduling problem with time windows. *European Journal of Operational Research*, 66(3):331–340, 1993.

Robert C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

Dabbala Raj Reddy. Speech understanding systems: Summary of results of the five-year research effort at Carnegie-Mellon University. Technical report, Carnegie Mellon University, 1977.

Andrea Saltelli. Making best use of model evaluations to compute sensitivity indices. *Computer Physics Communications*, 145(2):280–297, 2002.

Kate Smith. Neural networks for combinatorial optimization: A review of more than a decade of research. *INFORMS Journal on Computing*, 11(1):15–34, 1999.

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*, pp. 5998–6008, 2017.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *arXiv preprint arXiv:1506.03134*, 2015.

Chris Voudouris and Edward Tsang. Partial constraint satisfaction problems and guided local search. Technical report, Department of Computer Science, University of Essex, 1996.

Christos Voudouris and Edward Tsang. Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, 113(2):469–499, 1999.

Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. Learning improvement heuristics for solving routing problems. *arXiv preprint arXiv:1912.05784*, 2020.

A IMPLEMENTATION DETAILS

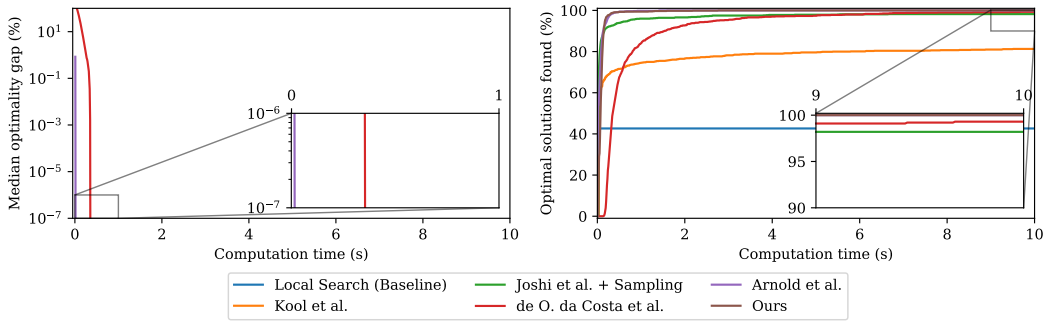
We train our model on sets of one hundred thousand 20, 50, and 100-node 2D Euclidean TSP instances. We generate all instances by uniform-randomly sampling node locations in the unit square $[0, 1]^2$. We calculate the target output, the regret of including each edge in the solution, according to eq. (4). We use Concorde to find an optimal solution and LKH-3 with 100 trials and 10 runs to solve find an optimal solution with a given edge fixed. We empirically validate that this configuration for LKH-3 is sufficient to find optimal solutions for problems with 100 nodes or less. We scale the input features and target outputs to a range of $[0, 1]$.

We train our model by minimizing the l^2 loss to the target output using the Adam optimizer (Kingma & Ba, 2014) and an exponentially decaying learning rate $\gamma = 10^{-3} \times (0.99)^e$, where e is the epoch. We train the model for 100 epochs with early stopping. We use a training batch size $B = 32$ for the 20 and 50-node training sets, and a training batch size of $B = 15$ for the 100-node training set due to limited GPU memory. Our model uses $T = 3$ message passing layers and our GLS algorithm uses $K = 20$ perturbation moves. Our approach is implemented using Python 3.8.11, PyTorch 1.9.0 (Paszke et al., 2019), DGL 0.7.1 (Wang et al., 2019), and CUDA 10.2, and is open-source.

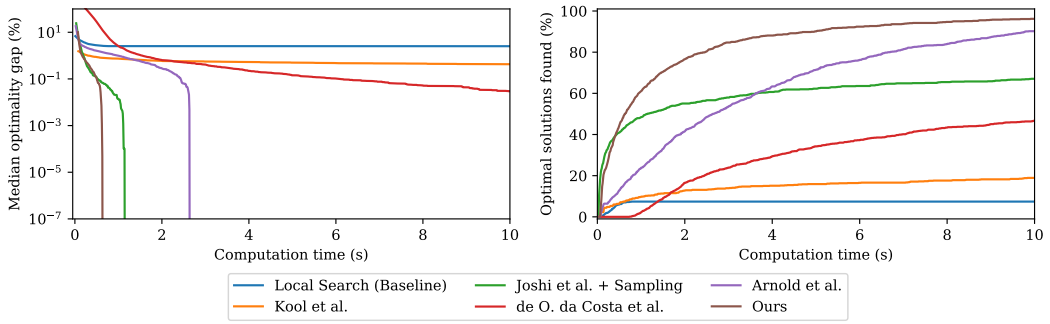
We conduct all computational experiments on a virtual machine with three cores of an Intel Xeon E5-2650 v4 processor, 24 GB of RAM, and a Tesla P100 GPU with 16 GB of VRAM.

B PERFORMANCE AS A FUNCTION OF TIME

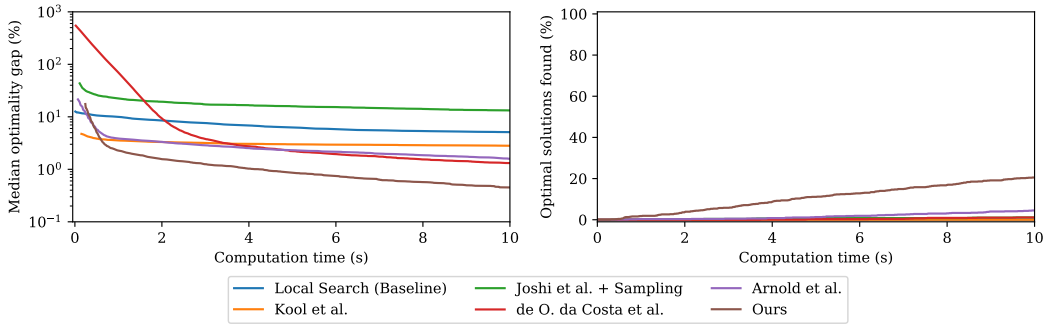
We evaluate the solution quality, in terms of median optimality gap and number of problems solved optimally, as a function of computation time rather than at the computation time limit. Figure 4 depicts this for our approach and several recent approaches, for models trained and evaluated on 20, 50, and 100-node problem instances. Computation time required to compute input features, evaluate a model, or to construct an initial solution is visible as a gap between the trace and the vertical axis, and is especially noticeable in Figure 4c. Note that in Figure 4a, the traces for the median optimality gap are not visible for some methods as these methods find many optimal solutions almost instantly. This is reflected in the plot depicting the number of optimal solutions found as a function of computation time. Figure 5 depicts the performance of our approach and the learning-based approaches when generalizing from smaller problems to larger ones.



(a) TSP20

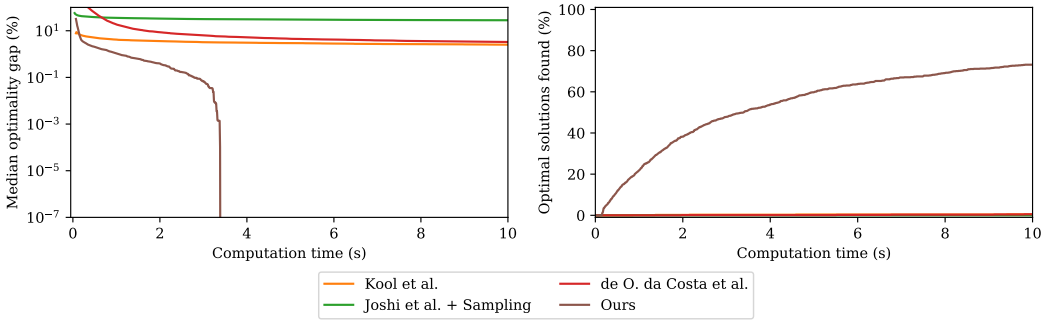


(b) TSP50

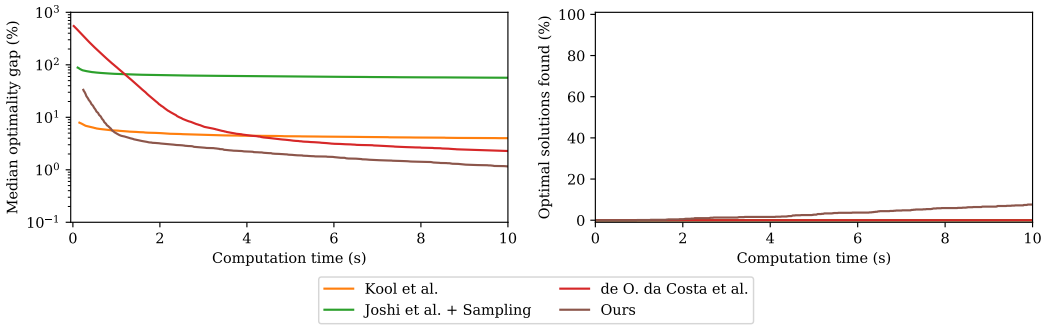


(c) TSP100

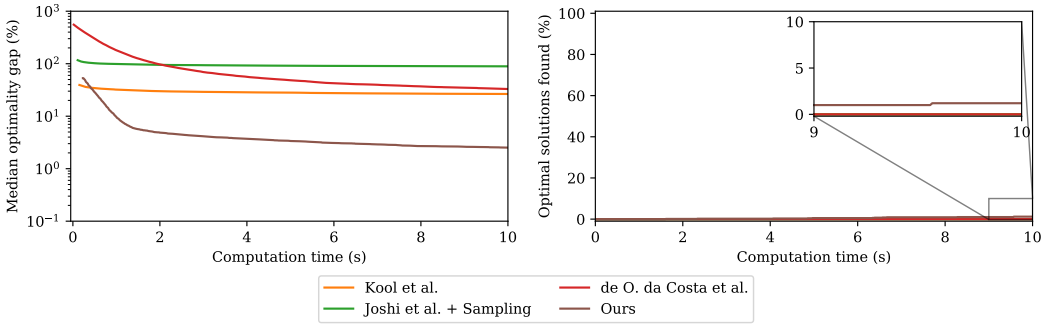
Figure 4: Solution quality, in terms of median optimality gap and percentage of optimally solved problems, as a function of computation time for three increasingly difficult problem sets, demonstrating that our method converges to optimal solutions at a faster rate than several benchmarks.



(a) Models trained on 20-node TSPs solving the TSP50 problem set.



(b) Models trained on 50-node TSPs solving the TSP100 problem set.



(c) Models trained on 20-node TSPs solving the TSP100 problem set.

Figure 5: Solution quality, in terms of median optimality gap and percentage of optimally solved problems, as a function of computation time for three increasingly difficult problem sets, demonstrating that our method is able to generalize from smaller to larger problems better than several learning-based benchmarks.

C ANALYSIS OF ADDITIONAL INPUT FEATURES

We evaluate the effect of adding additional features to our regret approximation model. While more features can help produce better predictions, they come at the cost of additional computation time. We consider a total of ten additional features, described in Table 3. We use a Gaussian process-based surrogate model to conduct global sensitivity analysis (Gratiet et al., 2016) of the input features on the validation loss of the model after training. We assume that better regret predictions will equate to better guidance by the GLS algorithm, ultimately resulting in better performance to the problem sets.

Name	Description	Complexity
Node width	Perpendicular distance from a node to a line from the depot node through the centroid of the nodes (Arnold & Sörensen, 2019b).	$\Theta(n)$
Edge width	Absolute difference between the width of two nodes.	$\Theta(n^2)$
Depot weight	Weight from a node to the depot node.	$\Theta(n)$
Neighbor rank	k where node j is the k-nearest neighbor of i. There are two features for a given edge, the rank of i w.r.t. j and the rank of j w.r.t. i.	$\Theta(n^2)$
30%-NN graph	If the edge is part of the k-nearest neighbor graph, where k is $0.3n$.	$\Theta(n^2)$
20%-NN graph	If the edge is part of the k-nearest neighbor graph, where k is $0.2n$.	$\Theta(n^2)$
10%-NN graph	If the edge is part of the k-nearest neighbor graph, where k is $0.1n$.	$\Theta(n^2)$
MST	If the edge is part of the minimum spanning tree, calculated using Prim's algorithm (Prim, 1957).	$\Theta(n^2)$
NN solution	If the edge is part of solution constructed by the nearest neighbor heuristic.	$\Theta(n)$
NI solution	If the edge is part of solution constructed by the nearest insertion heuristic.	$\Theta(n^3)$

Table 3: Summary of additional input features evaluated. While more features can help produce better predictions, and thus better guidance for the GLS algorithm, they come at the cost of additional computation time.

We semi-randomly sample 100 input feature sets using Latin hypercube sampling (McKay et al., 1979). We train a model using each of these feature sets for 35 epochs without early stopping and record the final validation loss. Using these results, we fit a Gaussian process to emulate the mapping between the feature set $F = \{f_0, f_1, \ldots, f_{10}\}$, and the validation loss l achieved by our model after training, where f_n indicates whether or not a feature is used. We then compute Monte-Carlo estimates of the main and total effects of each input feature on the model's validation loss (Saltelli, 2002). Our implementation uses Emukit (Paleyes et al., 2019). Figure 6 depicts the estimated total effect for each input feature. Edge weight is the most important feature, followed by neighbor rank, depot weight, edge width, and node width. The remaining features have little to no importance.

We train a model using these top five features on 20-node problems and compare its performance to the equivalent model using edge weight as the only feature when solving the 20, 50, and 100-node problem sets. The performance of both models at the computation time limit is shown in Table 4. While the model using additional input features produces slightly more accurate regret predictions, any benefit is cancelled out by the additional time required to compute these features. Note that the results are slightly different from those reported in Table 2 due to different training hyperparameters.

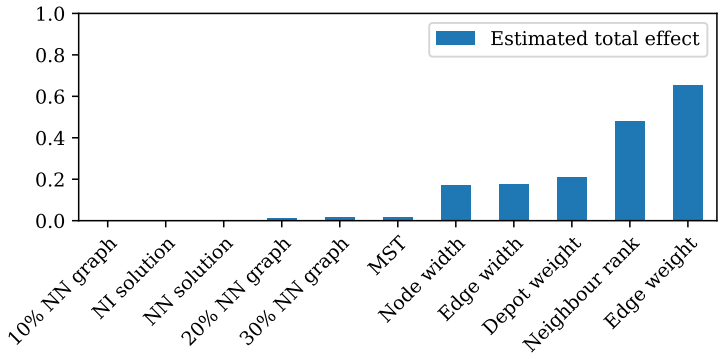


Figure 6: The estimated total effect of edge weight and ten additional features on model validation loss after training based on 100 randomly-sampled feature sets, using a global sensitivity analysis method based on Gaussian process surrogate modeling. A small effect implies that the feature is not important when predicting regret, and vice-versa.

Model	Problem	Method	Optimality gap (%) Median	Optimality gap (%) IQR	Optimal solutions (%)
TSP20	TSP20	**Edge weight only**	**0.0000**	**0.0000**	100.0000
		Top 5 features	0.0000	0.0000	99.8999
TSP20	TSP50	Edge weight only	0.0000	0.0000	82.0821
		Top 5 features	**0.0000**	**0.0000**	**83.1832**
TSP20	TSP100	**Edge weight only**	**1.9454**	**1.9661**	**2.2022**
		Top 5 features	2.0625	2.0009	2.2022

Table 4: Comparison of our method using a model using additional features and a model using edge weight alone. Both models are trained on 20-node problems and evaluated on the 20, 50, and 100-node problem sets. The median and inter-quartile range of the gap between the best and optimal solutions, and the percentage of optimally solved problems are reported after 10 seconds of computation time per instance. While the model using additional input features produces slightly more accurate regret predictions, any benefit is canceled out by the additional time required to compute these features.