Select and Optimize: Learning to solve large-scale TSP instances

Hanni Cheng **Haosi Zheng** **Ya Cong** **Weihao Jiang** **Shiliang Pu**
Hikvision Research Institute

Abstract

Learning-based algorithms to solve TSP are getting popular in recent years, but most existing works cannot solve very large-scale TSP instances within a limited time. To solve this problem, this paper introduces a creative and distinctive method to select and locally optimize subparts of a solution. Concretely, we design a novel framework to generalize a small-scale selector-and-optimizer network to large-scale TSP instances by iteratively selecting while optimizing one sub-problem. At each iteration, the running time of sub-problem sampling and selection is significantly reduced due to the full use of parallel computing. Our neural model is well-designed to exploit the characteristics of the sub-problems. Furthermore, we introduce a trick called destroy-and-repair to avoid the local minimum of the iterative algorithm from a global perspective. Extensive experiments show that our method accelerates state-of-the-art learning-based algorithm more than 2x while achieving better solution quality on large-scale TSP instances ranging in size from 200 to 20,000.

1 Introduction

Combinatorial Optimization (CO) aims to find the optimal solutions to optimization problems under integer constraints. One of the well-known examples is Traveling Salesmen Problem (TSP), which has been a most extensively studied NP-hard problem in the Operations Research (OR) community, with applications in transportation, logistics, and automation (Lenstra and Kan, 1974).

Nowadays, many traditional algorithms have already been proposed to solve it. Among these TSP algorithms, the Concorde TSP solver (Applegate et al., 2007) outperforms others at relatively small scales through linear programming with carefully handcrafted heuristics, while heuristics-based methods (Helsgaun, 2017) are capable of obtaining near-optimal solutions for instances with millions of cities. However, well-defined rules that rely on expert knowledge are always required, which makes these algorithms difficult to generalize to other CO problems. With the rapid development of computing power, deep learning techniques have been widely applied to images, text, videos, etc (Krizhevsky et al., 2012; Sutskever et al., 2014; Vaswani et al., 2017). Motivated by Pointer Network (Vinyals et al., 2015), a series of learning-based algorithms have also been raised to solve TSP instances, showing competitive performance on small-scale TSP instances compared with heuristics-based method while significantly reducing computation time (Khalil et al., 2017; Bello et al., 2017; Deudon et al., 2018; Kool et al., 2019; Joshi et al., 2019; Ma et al., 2019; Kwon et al., 2020).

However, learning-based methods for large-scale TSP instances have not yet achieved adequate performance. On the one hand, training a network for extremely large-scale problems faces a big challenge since it is nearly impossible to obtain abundant labels of large-scale instances within a limited time; on the other hand, the small-scale models can hardly generalize to large-scale instances due to the distributions varying from large-scale TSP instances to relatively smaller ones, which has been confirmed by Fu et al. (2021) and Joshi et al. (2021). As a result, the scaling of learned networks for CO problems is becoming popular in recent years. One possible way to solve this problem is *divide and conquer*: decomposing large-scale TSP instances into smaller ones which can be adequately solved by learning-based models and then combining the solutions of small-scale ones. Following this idea, Fu et al. (2021) trained a small-scale model which could be repeatedly used to build heat maps for TSP instances of arbitrarily large size and fed the heat maps into a reinforcement learning approach to guide the search. Unfortunately, it cannot solve an extremely large problem in a reasonable time because of the time-consuming search process of Monte Carlo Tree Search (MCTS). Specifically, it cannot get a high-quality solution of a 10000 instance within 15 minutes, which will not be acceptable in some real-time routing scenarios. Our work also follows the decompose-and-combine idea, but

Proceedings of the 26th International Conference on Artificial Intelligence and Statistics (AISTATS) 2023, Valencia, Spain. PMLR: Volume 206.

differs from Fu et al. (2021), we iteratively improve sub-parts of a solution with a well-trained small-scale model. Since we circumvent the global search part of large-scale problems, the time cost of our algorithms is significantly reduced.

In this work, we propose a creative and distinctive framework to **Select and Optimize** sub-parts of a solution. Concretely, we train a selector-and-optimizer network to iteratively select and locally optimize sub-paths of consecutive nodes in order to limit the selection space size to be linear to the problem size. Furthermore, another destroy-and-repair method (Pisinger and Røpke, 2018) is proposed to avoid the local minimum of the iterative algorithm. Overall, the main contributions are summarized as follows:

- We introduce a novel framework to generalize a small-scale selector-and-optimizer network to large-scale TSP instances by iteratively selecting and optimizing one sub-problem. At each iteration, sub-problems are sampled from the complete solution and the most promising sub-problem is then selected and optimized by our well-designed network with nearly $O(1)$ time complexity.
- We extend the classical Transformer-style Encoder-Decoder architecture to better adapt to the specifics of sub-problems. Incorporating the symmetry into policy gradient and inference stage, our design obtains superior *select-and-optimize* performance.
- Based on the result of our iteration framework, a destroy-and-repair method is proposed to further improve the solution quality. We devise special destroy and repair operators to destruct long connections of current solutions and fix them in a heuristic way to escape the local minima.

2 Related Work

2.1 Neural Methods for TSP

Owing to the highly structured nature of CO problems, deep-learning methods become a promising direction to solve them. Learning-based methods can be categorized into **Supervised Learning (SL)** methods and **Reinforcement Learning (RL)** methods, as discussed in the following two parts.

Supervised learning methods attempt to learn specific patterns or policies through optimal solution labels provided by conventional solvers. Pointer Network (Vinyals et al., 2015) is a sequence-to-sequence model equipped with an attention mechanism (Bahdanau et al., 2015), it constructs a solution through step-by-step decoding, while the training process is guided by expert solutions from Concorde. Joshi et al. (2019) design a deep Graph Convolutional Network (GCN) to build efficient TSP graph representations, the neural network outputs tours in a non-autoregressive manner via beam search. Kool et al. (2022) extract information from supervised pre-trained GCN as learned neural heuristics, which helps Dynamic Programming algorithms search more efficiently. Recently, Hudson et al. (2021) present a hybrid data-driven approach combining Graph Neural Networks and Guided Local Search, which converges to optimal solutions at a faster rate than three recent learning-based TSP algorithms.

Since optimal solution labels of TSP are unavailable to instantly access, reinforcement learning methods that do not need optimal labels are getting popular recently. Bello et al. (2017) present a framework integrating actor-critic reinforcement learning into LSTM-based Pointer Network, which achieves close to optimal results on TSP and Knap-Sack. Khalil et al. (2017) combine deep graph embedding with reinforcement learning and demonstrate the effectiveness of the proposed framework in learning greedy heuristics. Kool et al. (2019) specifically design an Encoder-Decoder architecture with an attention mechanism which is called Attention Model (AM), and the AM is trained using REINFORCE with a greedy rollout baseline. Ma et al. (2019) propose a Graph Pointer Network framework which efficiently solves larger-scale TSP and other constrained CO problems like TSPTW. Kwon et al. (2020) train the same Attention Model with a shared baseline policy gradient algorithm, this algorithm guides itself towards the optima by exploiting the symmetry of CO problems. Some other researches focus on improvement-based methods which aim at improving sub-optimal solutions with different schemes. da Costa et al. (2020) and Wu et al. (2021) both present improvement-based learning frameworks, which train deep RL to automatically discover better improvement policies. Ma et al. (2021) devise a novel Dual-Aspect Collaborative Transformer to learn embeddings for the node and positional features separately and capture the circularity of routing problems. Moreover, they incorporate curriculum learning into Proximal Policy Optimization (PPO) algorithm to learn improvement heuristics.

Some recent works have investigated generalizing TSP instances to different distributions by adding extra modules apart from SL and RL methods mentioned above. Wang et al. (2021); Geisler et al. (2021); Zhang et al. (2022) attempt to solve generalization problems with adversarial learning techniques, Boffa et al. (2022) show that augmenting the structural representation of problems with distance encoding is promising to enhance the encoder's representation power and generalize to "out of the box" problems. Jiang et al. (2022) study group distributionally robust optimization to improve the cross-distribution generalization ability.

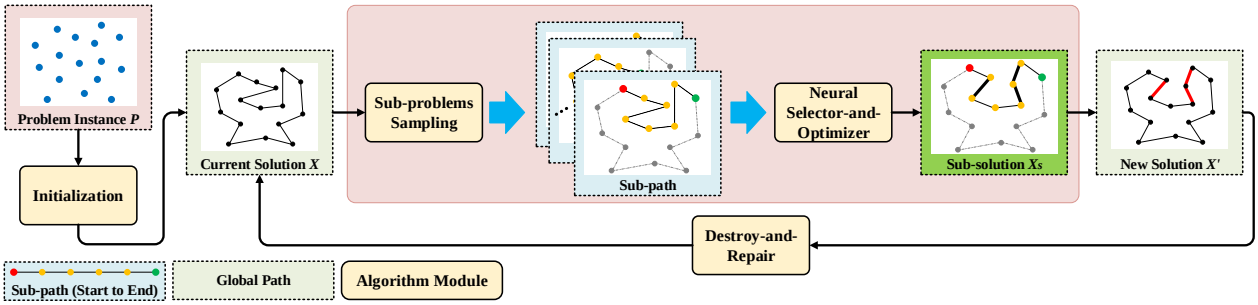


Figure 1: Our *select-and-optimize* iterative framework: At each step, batch sub-problems are sampled and then fed into our selector-and-optimizer network. We select the most promising sub-problem and transform it into a new sub-solution X_s by our network. The destroy-and-repair method is applied on the new solution X' at a certain updating interval.

2.2 Scaling up for CO Problems

Although learning-based algorithms achieve competitive performance on small-scale TSP instances, they suffer from generalizing to large-scale ones, namely most learning-based algorithms fail to obtain high-quality solutions on large-scale instances. Therefore, it is still a challenge to solve large-scale TSP instances within a reasonable time. Fu et al. (2021) manage to smoothly generalize a small-scale model to large-scale cases by training GCN which helps to build heat maps with a series of techniques. However, MCTS consumes too much time that one TSP instance of 10000 cannot be solved within 15 minutes, despite it already running much faster than heuristic methods. Ouyang et al. (2021) combine equivalence, local search, and stochastic curriculum learning techniques to directly improve generalization ability on TSP instances up to 10000, but its performance is inferior to Fu et al. (2021). Qiu et al. (2022) introduce a compact continuous space to parameterize the underlying distribution of candidate solutions and further propose a meta-learning framework over CO problem instances to enable effective initialization of model parameters in the fine-tuning stage. Experiments in Qiu et al. (2022)show that their proposed DIMES outperforms strong baselines among DRL-based solvers on TSP instances of 500, 1000 and 10000, but the time to obtain the solution is far longer than that of Fu et al. (2021). Other research like VSR-LKH (Zheng et al., 2021) and Neuro-LKH (Xin et al., 2021) replace inflexible part of LKH3 (Helsgaun, 2017) with learning-based methods to improve the robustness of original heuristic methods, but they have no advantages in time cost. As for other CO problems, Li et al. (2021) propose a learning-augmented framework to iteratively improve a large-scale CVRP solution, at each step a well-designed delegate network identifies an appropriate sub-problem to solve in a black-box way, which offers a 1.5x to 2x speedup over heuristics eventually. As the homochronous work of Li et al. (2021), Zong et al. (2022) solve large-scale VRPs by RL-based hierarchical framework named Rewriting-by-Generating (RBG) and show significant advantage on both solution quality and solving speed to other baselines. Ahn et al. (2020) propose an iterative scheme called *learning-what-to-defer* for a maximum independent set. It can be interpreted as prioritizing the easy decisions to be made first and then simplifying the difficult ones by eliminating the source of uncertainties, which also leads to a remarkable speedup.

Our framework for scaling up TSP is motivated by POPMUSIC (Taillard and Voß, 2002), which also follows the principle of *divide and conquer*. It comes up with the idea that locally optimizing sub-parts of an available solution brings global improvement. POPMUSIC has been widely applied in map labeling (Laurent et al., 2009) and p-median clustering (Taillard, 2003). In this paper, we take full advantage of the generalization capability of learning-based methods and parallel computing technology of the servers to accelerate our large-scale TSP algorithm. In particular, a well-designed *select-and-optimize* framework is leveraged to solve large-scale problems with considerable acceleration, while another destroy-and-repair method is raised to further improve the solution quality from a global perspective.

3 Methods

3.1 Preliminaries

We focus on TSP in 2D Euclidean space. Given nodes $1, ..., N$, $\mathbf{X} = \{x_i\}_{i=1}^{N}$ where the two-dimensional coordinate of the ith node can be described as $x_i \in [0, 1]^2$ without loss of generality. The objective of TSP is to find the shortest feasible route that visits each node exactly once so that the solution is a permutation of N nodes $\pi = [\pi_1, ..., \pi_N]$ that minimizes the total tour length. The length of a tour is defined as :

$$L(\pi) = \|x_{\pi_N} - x_{\pi_1}\|_2 + \sum_{i=1}^{N-1} \|x_{\pi_i} - x_{\pi_{i+1}}\|_2 \qquad (1)$$

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

3.2 Framework

The overview of our framework is illustrated in Figure 1. At the beginning of the framework, an initial solution is constructed by our innovative initialization method, which is an extension of initialization algorithm in Taillard and Helsgaun (2019), more details are included in Algorithm 1. At each iterative step, a number of sub-problems are sampled from the incumbent solution X. Then all sub-problems will be fed into our neural model in parallel and the sub-problem X_s with the largest improvement will be selected and optimized to update the current solution. A destroy-and-repair method is proposed to resist to local optimum at a specific frequency δ_d. The total iterative step is T. In the following sections, we will introduce sub-problem sampling and selection, our neural selector-and-optimizer architecture, and the destroy-and-repair method in detail.

3.3 Sub-problem Sampling

To optimize sub-problem iteratively, the sub-problem candidates should first be extracted from the large-scale TSP instance. Alternatively, we can optimize the connections inside a group of nodes that are the closest to a given node, but the cardinality of the selected space will be $O(N^2)$ because identifying such a group would take a computational effort in $O(N)$, which is prohibitive for very large instances. Thus we further restrict the selection space by leveraging the positions of the nodes in the complete solution since many combinatorial optimizations problems have inherent spatial locality. Specifically, we propose to optimize sub-paths of r consecutive nodes, and r is a relatively small number. In this way, if the sub-problem size we set is r, a tour of total N nodes can be considered as N sub-paths which contains r nodes, denoted as $\mathbf{S} = \{s_i\}_{i=1}^{N}$, sub-path s_i contains a series of consecutive nodes $i{+}1$, $i{+}2$, ..., $i + r$, which can easily be identified and located in a complete solution. The total computational effort is then reduced to $O(N)$ or even nearly $O(1)$ with the aid of parallel computing.

3.4 Sub-problem Selection

After N sub-problem candidates of size r are sampled from a current solution, only one sub-problem will be selected and optimized at each iteration. A neural selector-and-optimizer is trained to simultaneously select and optimize an appropriate sub-problem since many learning-based approaches have the ability to obtain high-quality solutions of small-scale TSP instances with fast speed, as shown in Figure 1. Specifically, the total N sub-problems are fed into the neural solver concurrently to obtain N new sub-costs. According to the sub-costs gap before and after the optimization, the most valuable sub-problem is then selected based on the immediate improvement. Note that only the sub-problem we select is optimized into a new sub-path, while the other sub-problems remain unchanged though their optimized sub-paths have already been calculated in the last step. Optimizing all the sub-problems concurrently seems no improvement comparing to just optimizing the best one according to our experiments, probably because the sub-problems have strong influences on each other.

During the iteration, a hill-climbing procedure is adopted, which means if the most valuable sub-problem cannot get any improvement after optimization, the incumbent solution will remain the same. In this way, we can avoid making the objective worse. Note that we only consider a one-step improvement, thus the solution tends to be trapped in the local optima, later we will introduce a destroy-and-repair method to alleviate this issue.

3.5 Neural selector-and-optimizer

In this section, we present our well-designed **neural selector-and-optimizer** that can select the most promising sub-problem and optimize it. The sub-problem is a little different from the original TSP. Firstly, the head and the tail of a sub-problem must be fixed to ensure that local improvement equals global improvement. Secondly, the head and the tail have no connections, namely the solution of a sub-problem is a sequence instead of a cycle. We define the sub-problem s as a sequence with r nodes, each node's coordinate is $x_i (i = 1, .., r)$ and the solution is $\pi = (\pi_1, .., \pi_r)$. Due to the aforementioned characteristics, the unique constraint for sub-problem is $\pi_1 = 1, \pi_r = r, \pi_t \in \{2, ..., r-1\}$ and $\pi_t \neq \pi_{t'} \forall t \neq t'$. Our neural selector-and-optimizer models the policy as $p_\theta(\pi|s)$, the optimization objective is to minimize the traveling tour length $L(\pi) = \sum_{i=1}^{r-1} \left\| x_{\pi_{i+1}} - x_{\pi_i} \right\|_2$.

To quickly obtain high-quality solutions of the sub-problems in parallel, we extend Transformer-style **Encoder-Decoder** architecture and train it with a modified **Policy Gradient** algorithm to handle this problem, which will be discussed in the following two sub-sections.

3.5.1 Network architecture

We inherit the Transformer-style Encoder-Decoder architecture from Kool et al. (2019) to solve TSP sub-problem since the Multi-Head Attention (MHA) structure of the encoder can be leveraged to encode the coordinates and the output of Decoder is exactly a sequence, which is consistent with our objective.

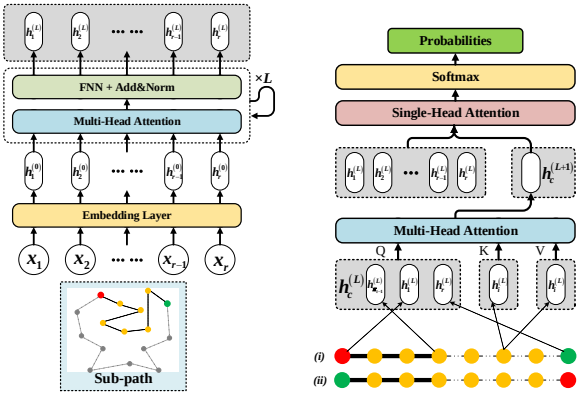
In the light of the characteristics of sub-problems, the original network is modified to better adapt to solving sub-problems. The Encoder architecture remains the same, as illustrated in Figure 2(a), which projects the 2-dimensional input feature x_i into d_h-dimensional vector $h_i^{(0)}$ and then followed by L stacked attention layers. As for Decoder, it is supposed to observe both the head and the tail informa-

tion of the route. Thus the context embedding $h_c^{(L)}$ should consist of three parts: the node embedding visited in the last time step $h_{\pi_t}^{(L)}$, the head embedding $h_1^{(L)}$, and the tail embedding $h_r^{(L)}$.

The decoding process at time t is as below:
a. Generate context embedding $h_c^{(L)} = [h_{\pi_t}^{(L)}, h_1^{(L)}, h_r^{(L)}]$
b. Compute the query, key, value: $q_c = W^Q h_c,\ k_i = W^K h_i,\ v_i = W^v h_i$ (L is omitted)
c. $q_c' = h_c^{(L+1)} = MHA(q, k, v)$
d. Compute the logits of candidate nodes with single-head attention, the tail and the visited nodes are masked:

$$u_{cj} = \begin{cases} C \cdot \tanh(\frac{q_c'^T k_j}{\sqrt{d_k}}), & if\ j \neq n\ and\ j \neq \pi_{t'}\ \forall t' < t \\ -\infty, & otherwise \end{cases} \tag{2}$$

where $d_k = \frac{d_h}{M}$, heads $M = 8$, clip value $C = 10$.
f. Calculate the final probability: $p_i = softmax(u_{ci})$



(a) Encoder (b) Decoder

Figure 2: The network architecture (a) Encoder: Encoding all the coordinate features of a sub-problem instance in one shot; (b) Decoder: Decoding from the two directions concurrently: from the head to the tail and the opposite direction.

3.5.2 Policy gradient with double optima

Compared with SL algorithms, RL models are guided by environmental reward signal so that it allows near-optimal solutions to be found without expert guidelines. As a result, we train our Encoder-Decoder with Policy Gradient (Williams, 2004), which is a classical reinforcement learning algorithm. The training loss function is defined as follows, where reward function $R(\pi) = -L(\pi)$, $b(s)$ denotes the average reward of the batch data.

$$\nabla L(\theta|s) = \mathrm{E}_{p_\theta(\pi|s)}[(R(\pi) - b(s))\nabla \log p_\theta(\pi|s)] \tag{3}$$

Inspired by POMO (Kwon et al., 2020), we exploit the symmetry of the sub-problem. It comes up with the idea that whether the route travels from the head to the tail or the opposite, both of the solutions are equally feasible. We perform different operations during training and testing to take advantage of this symmetry property. When training, our algorithm forces the network to produce the start point as the head or the tail, also called flip augmentation. While in the testing phase, decoding process is conducted from the two directions concurrently, the final inference solution is the better one among the two trajectories.

3.6 Destroy-and-repair method

The neural selector-and-optimizer mentioned above aims at iteratively locally optimize the whole problem, which may be trapped in local minimum due to the lack of a global perspective. Concretely, a situation may occur in our framework is that the route still contains very long connections after iterations, as illustrated in Figure 4. There is a possible reason that the nodes relatively far from each other may be the neighborhood in the initialized solution. What's worse, the sampling only considers consecutive nodes in a tour as sub-problems so that one node may never have the chance to connect to a nearby node in 2D Euclidean space. When the gap between the scale of the sub-problem and the original problem is too large, this situation may appear more frequently, which directly leads to the local optima instead of the global optima. Our proposed initialization method (Algorithm 1) helps to alleviate the problem, but another algorithm is needed as the iteration goes.

To tackle this issue, we propose a novel *destroy-and-repair* method to perturb the incumbent solution at a certain frequency during iteration, which was first proposed in Shaw (1998). In the destroy-and-repair phase, a destroy operator destructs part of the current solution while a repair operator rebuilds the destroyed solution. In particular, the destroy operator typically contains an element of stochasticity such that different parts of the solution are destroyed in every invocation of the method. In this situation, we aim to **destroy** the long connections and then **repair** the broken fragments to a complete route solution, as shown in Figure 3. In order to enhance the randomness of the proposed **destroy** operator, not only the long connections but also some others are destructed to make the sizes of all fragments resemble each other. After transforming the solution into d fragments, the *Lin–Kernighan* (Helsgaun, 2000) algorithm is leveraged to **repair** the broken fragments to form a feasible solution. The same hill-climbing strategy in select-and-optimize procedure is also kept in this stage, which means a new solution will only be accepted when its cost is lower than before. The destroy-and-repair method will only be conducted at a certain frequency in order to achieve a compromise between performance and runtime.

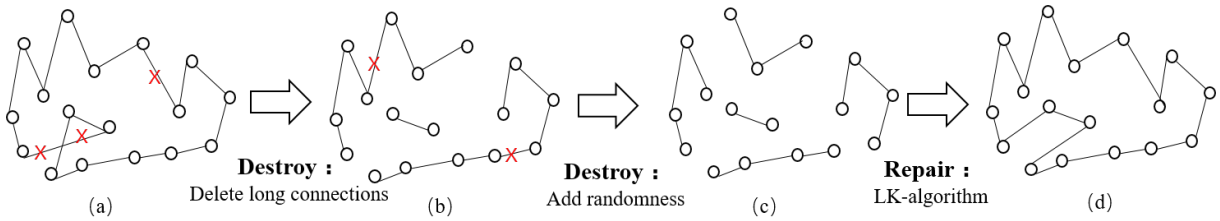


Figure 3: Illustration of destroy-and-repair method. (a) The original solution. (b) Destroy the longest connections in a solution. (c) Destroy some other connections to make the sizes of all fragments resemble each other. (d) Repair the d broken fragments into a feasible solution by Lin–Kernighan algorithm.

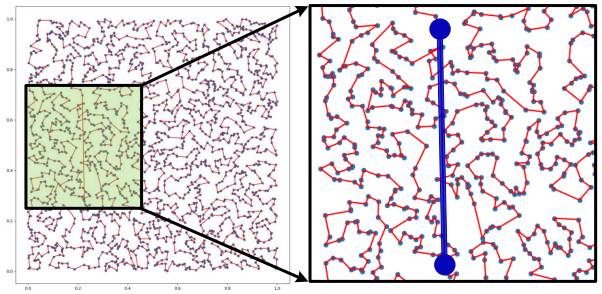


Figure 4: Bad long-connection case of a TSP2000 instance in our framework

4 Experiments and Analysis

In this section, we conduct two types of data sets experiments to evaluate our learning framework: generated data sets with various sizes and real-world benchmark TSPLIB (Reinelt, 1991). To verify the feasibility of our *select-and-optimize* iterative framework, comparison experiments are carried out to observe asymptotic behavior compared with heuristic selectors. Moreover, extensive ablation studies are conducted to exhibit the effectiveness of our neural network design and destroy-and-repair method.

4.1 Experimental setup

4.1.1 Data setup

Following the conventions, problem instances of our experiments are generated uniformly in the unit square. 1000 TSP instances are generated for training, respectively with $N = 200, 500, 1000$. Then an extension of initialization algorithm from Taillard and Helsgaun (2019) briefly initialize each problem instance. We train two neural selector-and-optimizer models of size $r = 50$ (*SO50* in brief) and $r = 100$ (*SO100* in brief) since experiments in Kool et al. (2019) and Kwon et al. (2020) show that neural methods can solve TSP instances of these two sizes with fast speed. For *SO50*, training sets are extracted from TSP200 and TSP500 instances, resulting in 1.4 millions samples with data augmentation. For *SO100*, training sets are extracted only from TSP1000 instances, in total 2.0 millions samples with data augmentation.

Pertaining to test data set, medium-scale and large-scale data groups are used to validate the effectiveness of our algorithm. For the medium-scale group, we adopt data set offered by Fu et al. (2021), 128 instances respectively with $N = 200, 500, 1000$. For the large-scale group, we generate 32 instances respectively with $N = 2000, 5000$ and 16 instances with $N = 10000$ by following the same rule as training sets.

4.1.2 Training hyperparameters

Training hyperparameters are set the same for the two neural models. The hidden dimension $d_h = 128$, the feed-forward sub-layer of each attention layer has a dimension $d_{ff} = 512$. The Adam optimizer is used with a learning rate of 0.0002 and weight decay of 10^{-6}. The policy network is trained with batch size 256, and 1000 epochs training takes approximately 16 hours for *SO50* and 40 hours for *SO100* on a single NVIDIA V100 GPU.

4.1.3 Algorithm parameters

The initialization algorithm for all the instances is the same as the training sets. As for iterative steps, we set $T = 50$ for medium-scale data, $T = 60, T = 80, T = 100$ for size of 2000, 5000, 10000 respectively, and $T = 100$ for TSPLIB instances. For the destroy-and-repair method parameters, updating interval $\delta_d = T/2$, separated fragments $d = 25$.

4.2 Performance on generated data set

There are two variants of our algorithm: Ours-SO100 iteratively optimizes the solution by only using *SO100* model; Ours-mixed re-optimizes the solution after *SO100* by using *SO50*, apparently this variant takes a little more time. Since the destroy-and-repair method doesn't have a significant effect on medium-scale data, we disable it in this experiment. We compare our algorithm with various types of solvers, (1) conventional solvers including Concorde (Applegate et al., 2007), LKH3 (Helsgaun, 2017), (2) learning-

based method including AM-greedy (Kool et al., 2019), POMO (Kwon et al., 2020), GCN-BS (Joshi et al., 2019), AttGCRN+MCTS (Fu et al., 2021). For these baselines, we just rerun the publicly available source codes. Table 1 demonstrates the results obtained by our algorithm on medium-scale data. Column 1 corresponds to the method name, and columns 2-4 respectively indicate average tour length, gap to the optimal solution (provided by Concorde), and runtime per instance (seconds). The runtime per instance metric is hard to compare due to various factors (e.g. Python v.s. C++, batch size, multi-processes). To ensure fair comparisons, we run C++ code on Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (e.g. Concorde, LKH3, MCTS) and Python code on one single NVIDIA Tesla V100 (e.g. POMO, AttGCRN). The runtime of AttGCRN-MCTS consists of the time for building heatmaps and running MCTS, while the runtime of our algorithm consists of initialization and iteratively solving sub-problems.

As shown in Table 1, our algorithm significantly outperforms all the learning-based methods both in terms of the quality of the solutions and the runtime. The inference time of AM-greedy and POMO is very short, but their performance deteriorates rapidly as the problem size increases. Compared to the SOTA on large-scale instances, AttGCRN-MCTS, our algorithm outstrips it within shorter than half of its runtime. Since our algorithm belongs to the learning-based category, we don't aim to strictly outperform the non-learning algorithms, but the gap can be further reduced within a little more inference time by Ours-mixed.

In Table 2, we only list results of LKH3, AM-greedy, POMO, AttGCRN-MCTS due to solving time and platform memory restriction. Note that we don't exploit the symmetry of sub-problems on TSP10000 due to GPU memory exceptions. Pertaining to large-scale data, the results are consistent with our design. The initialization method offers a rough solution to large-scale instances in a short time. Within a total runtime shorter than half of LKH3 and AttGCRN-MCTS, our algorithm achieves superior performance among all the learning-based methods.

Except for the results of TSP below 10000, we also test our algorithm on 8 extremely large-scale instances with 20000 vertexes to prove the generalization ability of our method, as shown in Table 3. POMO (Kwon et al., 2020) and AttGCRN-MCTS (Fu et al., 2021) fail to solve these instances due to GPU memory exceptions, thus we only compare our algorithm with AM-greedy (Kool et al., 2019) and LKH3 (Helsgaun, 2017). The performance of AM-greedy is extremely bad despite its lowest runtime. Our algorithm offers a 2x speedup over LKH3 while achieving competitive solution quality. The results demonstrate that our *select-and-optimize* design is an effective framework to solve large-scale TSP instances.

The results of Table 1, Table 2 and Table 3 demonstrate that our algorithm is capable of producing solutions close to LKH3 on large-scale TSP instances and the runtime is much shorter than LKH3 and the learning-based SOTA, and our work is the first neural method to test on TSP instances up to 20000.

4.3 Performance on real-world data set

To evaluate the generalization ability of our algorithm, Table 4 compares LKH3, OR-tools (Perron and Furnon, 2022) and AttGCRN-MCTS with our algorithm on large-scale instances of TSPLIB. The large-scale TSPLIB data are divided into two groups: 1000-2000 with 11 instances and 2000-6000 with 7 instances. Metrics include the average gap to the optimal solution and the total runtime. Note that the model tested in TSPLIB experiments is trained on the generated data set. As shown in Table 4, Ours-mixed significantly outperforms learning-based SOTA on all the data groups within a much shorter time. Ours-mixed offers 6x and 25x speedup over OR-tools on 1000-2000 group and 2000-6000 group respectively while achieving competitive solution qualities. LKH3 achieves best performance close to the optimum, but it costs too much time on some atypical instances such as *fl1400* and *fl1577* in TSPLIB.

Full detailed experimental results are provided in Table 7 of our appendix. Actually, AttGCRN-MCTS achieves better performance in some cases, but it can deteriorate badly on some atypical distribution data, e.g. *fl1400*. A possible reason is that the heat-map merging method of AttGCRN-MCTS is not robust to different distributions. However, our algorithm obtains more stable results across all kinds of data distribution with lower variance.

4.4 Sub-problem selector comparison

To verify the feasibility of our proposed iterative framework, we devise other two hand-crafted selector heuristics and another neural selector as baselines. The hand-crafted selectors are: 1) Random, selecting a sub-problem from batch sampling sub-problems randomly and uniformly; 2) Count, the route is divided into N/r buckets, the bucket with a smaller count has a larger probability to be selected, then a sub-problem is randomly sampled from the selected bucket. Another neural selector is called *nn-random*, which means randomly sampling a sub-problem from the promising candidates. While our neural selector is called *nn-argmax*, it selects the most promising sub-problem with the largest improvement. Note that though different selectors are utilized in this experiment, the sub-problem optimizer remains unchangeable as our neural model, which can also be replaced by other sub-solvers like LKH3.

As shown in Figure 5, we only report results on TSP500 and TSP2000 since similar patterns are observed in other cases. The x-axis is the iterative steps and the y-axis is the gap to the optimum. From the curves, we can con-

Method	TSP200			TSP500			TSP1000		
	Length	Gap	Time(s)	Length	Gap	Time(s)	Length	Gap	Time(s)
Concorde	10.7191	0.0000%	1.69	16.5458	0.0000%	22.90	23.1182	0.0000%	263.96
LKH3	10.7195	0.0040%	23.89	16.5463	0.0029%	73.12	23.1190	0.0036%	109.93
AM-greedy	11.9958	11.9105%	0.19	21.4619	29.7120%	0.55	33.5451	45.1025%	1.49
POMO	10.8923	1.6158%	0.05	20.5684	24.4206%	0.57	32.9014	42.3181%	4.08
GCN-BS	16.5331	54.2396%	0.91	30.1151	82.0105%	6.85	50.8119	119.7917%	38.18
AttGCRN-MCTS	10.8139	0.8844%	0.10+20	16.9655	2.5365%	0.2+50	23.8634	3.2238%	0.3+100
Ours-SO100	10.8145	0.8900%	1.00+8.59	16.9692	2.5590%	1.50+12.69	23.7827	2.8744%	2.94+21.74
Ours-mixed	10.7873	**0.6362%**	1.00+8.95	16.9431	**2.4012%**	1.50+13.50	23.7656	**2.8004%**	2.94+23.10

Table 1: Results on medium-scale data set, tested on 128 instances respectively with $N = 200, 500$ and 1000

Method	TSP2000			TSP5000			TSP10000		
	Length	Gap	Time(s)	Length	Gap	Time(s)	Length	Gap	Time(s)
LKH3	32.4507	0.0000%	157	50.8971	0.0000%	734	71.7752	0.0000%	1437
AM-greedy	52.9191	63.0753%	2	96.7435	90.0766%	4	153.4178	113.7504%	8
POMO	50.6465	56.0721%	5.71	88.1232	73.1399%	70.31	133.5866	86.1180%	609.75
AttGCRN-MCTS	33.5197	3.2942%	4+200	52.6867	3.5161%	20+500	74.4206	3.6856%	70+1000
Ours-SO100	33.3584	2.7971%	6.6+45.10	52.6080	3.3614%	20+180.32	74.3157	3.5395%	40+376.24
Ours-mixed	33.3456	**2.7577%**	6.6+48.38	52.5949	**3.3357%**	20+195.65	74.2993	**3.5166%**	40+417.60

Table 2: Results on large-scale data set, tested on 32 instances respectively with $N = 2000, 5000$ and 16 instances with $N = 10000$

Method	TSP20000		
	Length	Gap	Time(s)
LKH3	101.2861	0.0000%	4185.87
AM-greedy	244.1992	113.7504%	19.22
Ours-SO100	105.0575	4.2195%	185+1280
Ours-mixed	105.0344	**3.7101%**	185+1822

Table 3: Results on TSP20000

Method	1000-2000		2000-6000	
	Gap	Time(s)	Gap	Time(s)
LKH3	0.05%±0.07%	323	0.09%±0.14%	1098
OR-tools	5.74%±1.78%	285	**4.89%**±2.20%	3291
AttGCRN-MCTS	308.84%± 1011.71%	139	1579.53%± 2675.90%	393
Ours-mixed	**3.75%**±1.69%	45	5.46%±2.35%	125

Table 4: TSPLIB results

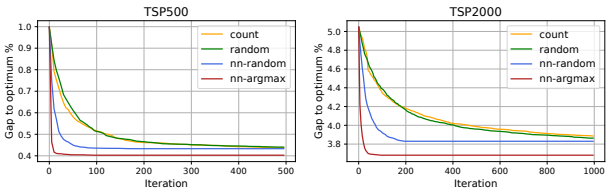


Figure 5: Selector comparison experiments

clude that all the selectors based on our iterative framework converge in a reasonable time, which indicates that our iterative framework works whatever the selector is. Furthermore, neural selectors converge significantly faster than hand-crafted heuristics. Among these selectors, *nn-argmax* exhibits the fastest converge speed and the best solution quality, which validates the effectiveness of our neural model's selecting capability.

4.5 Ablation study

To better understand how the different components affect our framework, ablation studies are conducted on all the synthetic data sets. For medium-scale data, only the specially designed components of the neural model are checked, including the tail embedding coupled with the query vector (corresponding to w/o tail) and the symmetry of sub-problems (corresponding to w/o symmetry). For large-scale data, destroy-and-repair method (corresponding to w/o DR) is also considered. Note that the time column excludes the runtime of the initialization.

4.5.1 About neural network

Pertaining to the ablation study about the neural network, we only report the results on TSP500 and TSP2000 due to the page limitation. The results from Table 5 indicate that every component plays an important role in solving sub-problems. In particular, adding the tail embedding into the query vector significantly boosts the performance of the neural network. Without exploiting the symmetry, the inference time can be reduced by sacrificing solution quality. It is worth noting that the inference time decreases more as the problem size increases.

Instance	Ours-SO100		w/o tail		w/o symmetry	
	Gap	Time(s)	Gap	Time(s)	Gap	Time(s)
TSP500	2.56%	13	3.17%	13	2.75%	11
TSP2000	2.79%	45	2.96%	45	2.83%	31

Table 5: Ablation study on TSP500 and TSP2000

Instance	Ours-SO100		w/o DR	
	Gap	Time(s)	Gap	Time(s)
TSP2000	2.7971%	45	2.8486%	39
TSP5000	3.3614%	180	3.3897%	156
TSP10000	3.5395%	376	3.5708%	305

Table 6: Ablation study about destroy-and-repair method

4.5.2 About destroy-and-repair method

Table 6 demonstrates the ablation study of destroy-and-repair (DR) method on the overall large-scale data. Clearly, DR module can ameliorate the performance of our algorithm plus a little more time. The results are consistent with our assumption that globally changing the structure of the solution route helps to be resistant to local minima.

5 Conclusion

As a well-studied combinatorial optimization problem, TSP has been widely applied in many scenarios of our real life, but how to solve large-scale TSP in a very limited time remains a problem. In this paper, an iterative framework to solve large-scale TSP instances is proposed with remarkable acceleration. Our *select-and-optimize* framework significantly reduces computation costs by dividing large-scale problems into smaller ones and solving the sub-problems with trained neural network iteratively, while further improving the quality of the solution by the destroy-and-repair module. The sub-problem sampling and selection strategy, the elaborate neural selector-and-optimizer network design, and the destroy-and-repair approach enable us to access high-quality solutions in a very limited time. Experimental results of the generated data and real-world data from TSPLIB show that our algorithm is able to produce highly competitive solutions compared with other learning-based TSP algorithms while taking less than half the runtime of them. Moreover, we change the selector in the iterative framework and empirically justify that whatever the selector is, iteratively selecting and optimizing a sub-problem is a feasible technique to solve large-scale problems.

We would like to address some options for future work. First, we have only experimented under the standard TSP instances, we will further explore the application in real-word scenarios with more practical constraints and large differences in data distribution. Second, we believe that following the principle of *divide and conquer*, the *select-and-optimize* iterative framework has the potential to solve other large-scale CO problems such as Capacitated Vehicle Routing Problem (CVRP) and Bin Packing Problem (BPP). Third, simulated annealing and other evolutionary algorithms combined with Machine Learning seem to alleviate the local optima problem we still face in *select-and-optimize* framework, which will be a very promising research direction.

References

Ahn, S., Seo, Y., and Shin, J. (2020). Learning what to defer for maximum independent sets. In *ICML*.

Applegate, D. L., Bixby, R. E., Chvátal, V., and Cook, W. J. (2007). The traveling salesman problem: A computational study.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. (2017). Neural combinatorial optimization with reinforcement learning. *ArXiv*, abs/1611.09940.

Boffa, M., Ben-Houidi, Z., Krolikowski, J., and Rossi, D. (2022). Neural combinatorial optimization beyond the tsp: Existing architectures under-represent graph structure. *ArXiv*, abs/2201.00668.

da Costa, P., Rhuggenaath, J., Zhang, Y., and Akcay, A. E. (2020). Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. *ArXiv*, abs/2004.01608.

Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., and Rousseau, L.-M. (2018). Learning heuristics for the tsp by policy gradient. In *CPAIOR*.

Fu, Z.-H., Qiu, K.-B., and Zha, H. (2021). Generalize a small pre-trained model to arbitrarily large tsp instances. In *AAAI*.

Geisler, S., Sommer, J., Schuchardt, J., Bojchevski, A., and Gunnemann, S. (2021). Generalization of neural combinatorial solvers through the lens of adversarial robustness. *ArXiv*, abs/2110.10942.

Helsgaun, K. (2000). An effective implementation of the lin-kernighan traveling salesman heuristic. *Eur. J. Oper. Res.*, 126:106–130.

Helsgaun, K. (2017). An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems: Technical report.

Hudson, B. H., Li, Q., Malencia, M., and Prorok, A. (2021). Graph neural network guided local search for the traveling salesperson problem. *ArXiv*, abs/2110.05291.

Jiang, Y., Wu, Y., Cao, Z., and Zhang, J. (2022). Learning to solve routing problems via distributionally robust optimization. In *AAAI*.

Joshi, C. K., Cappart, Q., Rousseau, L.-M., Laurent, T., and Bresson, X. (2021). Learning tsp requires rethinking generalization. In *CP*.

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

Khalil, E. B., Dai, H., Zhang, Y., Dilkina, B. N., and Song, L. (2017). Learning combinatorial optimization algorithms over graphs. In *NIPS*.

Kool, W., van Hoof, H., Gromicho, J. A. S., and Welling, M. (2022). Deep policy dynamic programming for vehicle routing problems. In *CPAIOR*.

Kool, W., van Hoof, H., and Welling, M. (2019). Attention, learn to solve routing problems! In *ICLR*.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25(2).

Kwon, Y.-D., Choo, J., Kim, B., Yoon, I., Min, S., and Gwon, Y. (2020). Pomo: Policy optimization with multiple optima for reinforcement learning. *ArXiv*, abs/2010.16011.

Laurent, M., Taillard, É. D., Ertz, O., Grin, F., Rappo, D., Roh, S.-J., and de Cheseaux, R. (2009). From point feature label placement to map labelling.

Lenstra, J. K. and Kan, A. H. G. R. (1974). Some simple applications of the travelling salesman problem. *Journal of the Operational Research Society*, 26:717–733.

Li, S., Yan, Z., and Wu, C. (2021). Learning to delegate for large-scale vehicle routing. In *NeurIPS*.

Ma, Q., Ge, S., He, D., Thaker, D. D., and Drori, I. (2019). Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. *ArXiv*, abs/1911.04936.

Ma, Y., Li, J., Cao, Z., Song, W., Zhang, L., Chen, Z., and Tang, J. (2021). Learning to iteratively solve routing problems with dual-aspect collaborative transformer. In *NeurIPS*.

Ouyang, W., Wang, Y., Weng, P., and Han, S. (2021). Generalization in deep rl for tsp problems via equivariance and local search. *ArXiv*, abs/2110.03595.

Perron, L. and Furnon, V. (2022). Or-tools.

Pisinger, D. and Røpke, S. (2018). Large neighborhood search. *Handbook of Metaheuristics*.

Qiu, R., Sun, Z., and Yang, Y. (2022). Dimes: A differentiable meta solver for combinatorial optimization problems. *ArXiv*, abs/2210.04123.

Reinelt, G. (1991). Tsplib - a traveling salesman problem library. *INFORMS J. Comput.*, 3:376–384.

Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. In *CP*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *NIPS*.

Taillard, É. D. (2003). Heuristic methods for large centroid clustering problems. *Journal of Heuristics*, 9:51–73.

Taillard, É. D. and Helsgaun, K. (2019). Popmusic for the travelling salesman problem. *Eur. J. Oper. Res.*, 272:420–429.

Taillard, É. D. and Voß, S. (2002). Popmusic ¡ª partial optimization metaheuristic under special intensification conditions.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer networks. In *NIPS*.

Wang, C., Yang, Y., Slumbers, O., Han, C., Guo, T., Zhang, H., and Wang, J. (2021). A game-theoretic approach for improving generalization ability of tsp solvers. *ArXiv*, abs/2110.15105.

Williams, R. J. (2004). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

Wu, Y., Song, W., Cao, Z., Zhang, J., and Lim, A. (2021). Learning improvement heuristics for solving routing problems.. *IEEE transactions on neural networks and learning systems*, PP.

Xin, L., Song, W., Cao, Z., and Zhang, J. (2021). Neurolkh: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem. *ArXiv*, abs/2110.07983.

Zhang, Z., Zhang, Z., Wang, X., and Zhu, W. (2022). Learning to solve travelling salesman problem with hardness-adaptive curriculum. In *AAAI*.

Zheng, J., He, K., Zhou, J., Jin, Y., and Li, C. (2021). Combining reinforcement learning with lin-kernighan-helsgaun algorithm for the traveling salesman problem. In *AAAI*.

Zong, Z., Wang, H., Wang, J., Zheng, M., and Li, Y. (2022). Rbg: Hierarchically solving large-scale routing problems in logistic systems via reinforcement learning. *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*.

Select and Optimize: Learning to solve large-scale TSP instances Supplementary Materials

A Solution Initialization

Since the initial solution plays an important role in iterative algorithms, we pay much attention to the solution initialization algorithm. da Costa et al. (2020); Wu et al. (2021); Ma et al. (2021) focus on small-scale TSP algorithms so that completely random solutions are enough for initialization. For large-scale instances, an acceptable solution should be obtained in a short time. Moreover, it should have the potential to be improved at each iteration. Li et al. (2021) generate a rough initial solution by partitioning it into disjoint subsets of nodes and briefly running the subsolver on each subset. Taillard and Helsgaun (2019) get an initial solution by local clustering optimization with a relatively low complexity, which is empirically convenient for their proposed iterative framework. In the following sections, we present an innovative algorithm inherited from Taillard and Helsgaun (2019) and conduct an initialization sensitivity experiment to demonstrate the effectiveness of our initialization method.

A.1 Initialization method

In the original POPMUSIC (Taillard and Helsgaun, 2019) initialization method line 1, uniformly sampling anchor nodes is an important step to form a base structure. Following the uniform principle, cluster centers are generated by K-means algorithm, then the nearest nodes to cluster centers are assigned as anchor nodes. With the aid of K-means approach, the initial tour becomes deterministic rather than random previously. Furthermore, we re-optimize sub-paths of 3 clusters rather than 2 to avoid long connections, which is stated in line 10-12 of Algorithm 1. Based on the same principle, the tour is split into N/Δ sections at the final step, and each section is re-optimized by LK algorithm. The whole modified method is illustrated in Algorithm 1. In the experimental part of the main text, we set $a = n^{0.56}$, $\Delta = N/4$ for medium-scale data and $\Delta = 500$ for large-scale data.

Algorithm 1 Initialization

Input: n nodes, distance function $d(i, j)$ between node i and node j
Parameter: the number of cluster centers a, the section length Δ
Output: TSP tour T

1: Generate a cluster centers by K-means algorithm
2: Select the nearest nodes to centers into sample set S
3: Build a LK-optimal tour T_s on S
4: $T = T_s$
5: **for** each node $c \notin S$ **do**
6: $\quad c_s = \arg\min d(s, c), (s \in S)$
7: $\quad$ Insert c just after c_s in T
8: **end for**
$\quad c = T_s[0]$
9: **while** $c \neq T_s[-1]$ **do**
10: $\quad$ Let c' be the node after the the node next to c in T_s
11: $\quad$ Optimise in T a sub-path between c and c' with LK algorithm
12: $\quad c \leftarrow c'$
13: **end while**
14: Split T into N/Δ sections
15: Optimise the N/Δ sections with LK algorithm

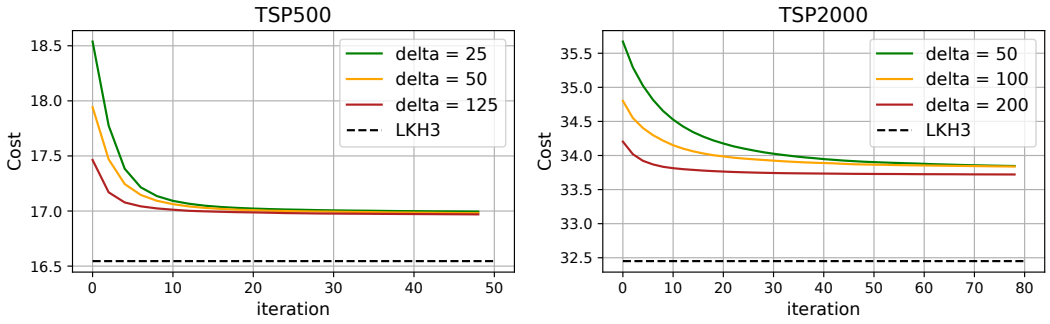


Figure 6: Initialization sensitivity experiments

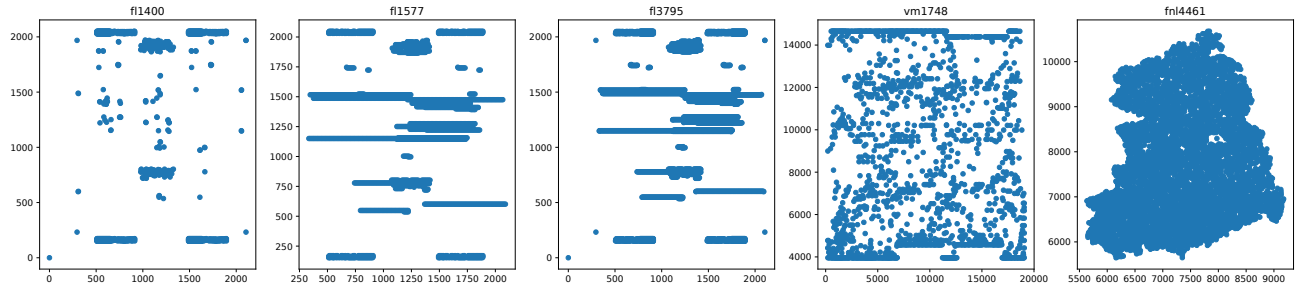


Figure 7: TSPLIB sample instances

A.2 Initialization sensitivity

The problem instance initialization is the first step of our framework. To study the sensitivity of our framework to initialization quality, different solutions with $\Delta = 25, 50, 125$ for TSP500 and $\Delta = 50, 100, 200$ for TSP2000 are additionally generated. Since smaller Δ means rougher division, the smaller Δ corresponds to the worse initial solution quality.

We speculate that whatever the quality of the initial solution, Algorithm 1 provides a convenient structure for being iteratively optimized. To validate this assumption, our method runs for an excessive amount of time until convergence. From Figure 6, we can conclude that our initial algorithm is well-suited for the iterative framework. Moreover, worse initialization corresponds to somewhat worse acceleration, but the initial gap can be significantly reduced through iterations. As introduced in the main text, our method is vulnerable to the local minima, therefore there is still a minor gap between convergence solution and global optima.

B Experimental Results

B.1 TSPLIB full results

Full results on large-scale TSPLIB instances are listed in Table 7. We compare our algorithm with a popular commercial solver OR-tools (Perron and Furnon, 2022), learning-based SOTA AttGCRN-MCTS (Fu et al., 2021) and a famous heuristic LKH3 (Helsgaun, 2017). The bold number indicates the best performance among learning-based methods and OR-tools. Clearly, Ours-SO100 achieves superior performance on most instances within a much shorter time than baselines. It is obvious that OR-tools suffers from a huge time budget and AttGCRN-MCTS suffers from unstable results, which means AttGCRN-MCTS fails to generalize to some special data distributions. LKH3 is capable of solving most instances at a favorable speed except for the instances starting with *fl* since the distributions vary a lot from other instances, as shown in Figure 7. The results indicate that heuristic methods also lack generalization ability.

B.2 Ablation study on selector-and-optimizer

Full ablation results of neural selector-and-optimizer are illustrated in Table 8. It can be concluded that integrating the tail embedding into the query vector is essential for solving sub-problems, and the characteristic of symmetry also helps to

Problem	Opt	Ours-mixed			OR-tools			AttGCRN-MCTS			LKH3		
		Length	Gap	Time(s)	Length	Gap	Time(s)	Length	Gap	Time(s)	Length	Gap	Time(s)
u1060	224094	**229053**	2.2132%	35	236208	5.4058%	172	230218	2.7327%	106	224094	0.0000%	18
vm1084	239297	244571	2.2041%	35	250085	4.5081%	192	**244221**	2.0576%	109	239297	0.0000%	21
pcb1173	56892	**58525**	2.8715%	38	59591	4.7446%	174	58588	2.9810%	117	56892	0.0000%	0.13
rl1304	252948	270048	6.7605%	42	278717	10.1875%	245	**259179**	2.4633%	130	253296	0.1375%	9
rl1323	270199	281583	4.2135%	42	284340	5.2335%	210	**276644**	2.3852%	133	270199	0.0000%	0.29
nrw1379	56638	**57564**	1.6363%	44	59182	4.4911%	228	57758	1.9774%	138	56643	0.0088%	10
fl1400	20127	**20521**	1.9614%	45	21583	7.2328%	258	696244	3359.2537%	140	20164	0.1838%	2107
fl1577	22249	23233	4.4260%	51	**23020**	3.4663%	267	24887	11.8567%	158	22262	0.0584%	1310
vm1758	336556	348692	3.6061%	53	355571	5.6499%	436	**342814**	1.8594%	175	336750	0.0576%	6
u1817	57201	**59254**	3.5905%	56	60774	6.2460%	438	60407	5.6047%	182	57287	0.1503%	29
rl1889	316536	331186	4.6284%	58	335676	6.0467%	518	**329426**	4.0722%	189	316549	0.0041%	43
d2103	80450	87141	8.3179%	71	**81813**	1.6944%	955	83665	3.9962%	212	80462	0.0149%	60
u2152	64253	**67408**	4.9113%	66	69784	8.6080%	794	67756	5.4518%	217	64310	0.0887%	49
pcb3038	137694	**141377**	2.6749%	93	144510	4.9503%	1417	7879443	5622.4301%	305	137700	0.0043%	25
fl3795	28772	30353	5.4982%	121	**29646**	3.0370%	2277	36574	27.1166%	381	28885	0.3927%	7427
fnl4461	182566	**187005**	2.4318%	139	190960	4.5976%	4164	9982482	5367.8757%	451	182571	0.0027%	37
rl5915	565530	610728	7.9922%	194	**600028**	6.1001%	7381	644331	13.9340%	594	566367	0.1480%	48
rl5934	556045	590189	6.1406%	196	**585236**	5.2497%	6051	644345	15.8800%	596	556143	0.0176%	42

Table 7: Large-scale TSPLIB results

Instance	Ours-SO100		w/o tail		w/o symmetry	
	Gap	Time	Gap	Time	Gap	Time
TSP200	0.8900%	9	1.6811%	9	1.2380%	8
TSP500	2.5590%	13	3.1682%	13	2.7548%	11
TSP1000	2.8744%	22	3.1456%	22	2.9911%	15
TSP2000	2.7971%	45	2.9611%	45	2.8310%	31
TSP5000	3.3614%	180	3.4933%	180	3.3799%	127
TSP10000	3.5395%	376	3.6795%	376	-	-

Table 8: Ablation study about neural network

boost the performance.