Solving the Min-Max Multiple Traveling Salesmen Problem via Learning-Based Path Generation and Optimal Splitting

Wen Wang[a], Xiangchen Wu[a], Liang Wang[a,*], Hao Hu[a], Xianping Tao[a] and Linghao Zhang[b]

[a]Nanjing University
[b]State Grid Sichuan Electric Power Research Institute

Abstract. This study addresses the Min-Max Multiple Traveling Salesmen Problem (m^3-TSP), which aims to coordinate tours for multiple salesmen such that the length of the longest tour is minimized. Due to its NP-hard nature, exact solvers become impractical under the assumption that $P \neq NP$. As a result, learning-based approaches have gained traction for their ability to rapidly generate high-quality approximate solutions. Among these, two-stage methods combine learning-based components with classical solvers, simplifying the learning objective. However, this decoupling often disrupts consistent optimization, potentially degrading solution quality. To address this issue, we propose a novel two-stage framework named **Generate-and-Split** (GaS), which integrates reinforcement learning (RL) with an optimal splitting algorithm in a joint training process. The splitting algorithm offers near-linear scalability with respect to the number of cities and guarantees optimal splitting in Euclidean space for any given path. To facilitate the joint optimization of the RL component with the algorithm, we adopt an LSTM-enhanced model architecture to address partial observability. Extensive experiments show that the proposed GaS framework significantly outperforms existing learning-based approaches in both solution quality and transferability.

1 Introduction

The multiple traveling salesmen problem (mTSP) generalizes the classical traveling salesman problem (TSP) by involving multiple salesmen who depart from a common depot, collectively visit all cities exactly once, and return to the depot. The mTSP is a versatile model applicable to a wide range of real-world problems [28, 3, 6]. Solving the mTSP requires generating valid tours for all salesmen while optimizing a specific objective. Two common objectives are the min-sum, which minimizes the total tour length, and the min-max, which minimizes the length of the longest individual tour. We study the min-max mTSP, abbreviated as m^3-TSP, which focuses on improving load balance and reducing worst-case latency.

There lack efficient, polynomial-time solutions to m^3-TSP because the problem is NP-hard [5], and as long as $P \neq NP$. Classic solvers, based on mathematical programming [4] or heuristic search [9], typically require a substantial amount of time to find a feasible solution. In recent years, RL-based methods [17] have received much research interest because they are promising in generating high-quality solutions efficiently. One series of RL-based methods, such as DAN [2], ScheduleNet [25], Equity-Transformer (Eqt for short) [27], and Dpn [35] perform end-to-end training by directly generating solutions to the original m^3-TSP and optimizing the process. Although end-to-end training ensures the searching direction for the optimal solution to the original problem, a reasonable problem decomposition, allowing RL to focus on a simpler part of the problem, is expected to reduce the learning complexity.

Unlike the above work, the other series of work propose to decompose the problem and adopt RL as a part of their solutions. These methods heuristically decompose the m^3-TSP into two interdependent subproblems: 1) *city-division*: dividing the cities into groups, each designated for a specific salesman; and 2) *path-generation*: generating a tour for each salesman to visit the cities within their assigned group. Following this framework, DisPN [12] employs RL to first partition the cities among the salesmen, then determines the optimal tour for each salesman by solving smaller-scale TSP instances with classic solvers. Differently, SplitNet [20] first uses classic solvers to generate an tour visiting all cities (by solving the TSP for a single salesman) and then learns to split this tour for the salesmen. By decomposing the problem, these approaches effectively simplify the learning process by focusing on city division and leaving path generation to other methods. However, solving the subproblems independently can lead to sub-optimal overall solutions compared to end-to-end methods.

Inspired by both lines of research, we propose the Generate-and-Split (GaS) framework, an RL-based approach that integrates problem decomposition with joint training to solve the m^3-TSP.

Firstly, given an instance of m^3-TSP, we train an RL-based **path generator** that generates a permutation of cities step-by-step, with each step selecting an unvisited city. Unlike end-to-end methods that address city division and path generation simultaneously by creating specific routes for each salesman, the path generator finds an path visiting all cities, which is then split into tours and assigned to individual salesmen—similar to SplitNet's approach. By focusing on learning for path generation, we simplify the problem, reducing it to one with a shorter decision horizon. In contrast to prior approaches such as DisPN and SplitNet, which address subproblems in isolation, the learning-based generator operates within a unified training process, aiming to produce paths that facilitate low-cost solutions through the utilization of the splitting algorithm. Secondly, we present a straightforward and efficient, optimal **splitting algo-**

* Corresponding Author. Email: wl@nju.edu.cn

rithm designed to split any specified path into tours. The splitting algorithm divides a sequence of length $n-1$ into m subsequences, with an objective of minimizing the maximum cost of the tours corresponding to the subsequences, where $n-1$ refers to the number of cities (excluding the depot), and m is the number of traveling salesmen. A brute-force method can solve the splitting problem with a prohibitive time complexity of $O(n^{m-1})$ by enumerating all possible split points. As a result, SplitNet adopts a learning-based approach to find effective splitting strategies in a computationally efficient manner. Unlike SplitNet, in this work, we show that *we can find optimal (without considering the float-pointing-error) splitting strategies with an efficient, deterministic algorithm, given that we define m^3-TSP in the Euclidean space*. More specifically, we propose a splitting algorithm that can approximate the optimal solution with an approximation ratio of $1+\epsilon$, $\forall \epsilon > 0$, with a time complexity of $O(n \log \frac{\hat{c}}{\epsilon})$ for any given path. $\hat{c}$ is the maximum possible cost, and ϵ represents the desired precision. Thirdly, since the splitting procedure is implemented subsequent to generating the entire path, essential information—such as the number of traveling salesmen who have finished their tours—is not accessible at the moment decisions are made. This situates the challenges of partial observation in the framework of RL. To address this, we model the task as a partially observable Markov decision process (POMDP) [15] and adopt an LSTM-based architecture [10] to mitigate the effects of partial observability. The use of recurrent neural networks (RNNs) to tackle partial observability dates back to the early development of deep RL [8], and has since been adopted in combinatorial optimization frameworks [1, 13], an idea we also draw upon in this work.

In summary, this work makes the following contributions:

- We propose the GaS framework for the m^3-TSP problem, which adopts the principle of problem decomposition. It integrates a path generator to tackle the more challenging aspects of the problem, and a deterministic splitting algorithm to efficiently address the remaining components.
- We design an LSTM-enhanced decoder to support joint learning together with the splitting algorithm. The splitting algorithm, though simple and efficient, plays a crucial role by providing effective reward signals for the RL-based path generator.
- We conduct extensive experiments to evaluate the solution quality and transferability of the proposed GaS framework. The results show that, compared to strong baselines such as Eqt [27] and Dpn [35], our approach achieves a statistically significant performance improvement ($p < 0.05$) and demonstrates better transferability in 90.7% of the out-of-distribution data.

2 Problem Formulation

The m^3-TSP is defined on a *complete* graph $G = (V, E)$, with $V = \{v_0, v_1, \cdots, v_{n-1}\}$ denotes the set of n nodes (which include a depot v_0, and $n-1$ cities from v_1 to v_{n-1}), and E denotes the set of edges. Following existing work [20], we place the nodes on a 2-dimensional Euclidean plane with (x_v, y_v) denotes the coordinate of a node $v \in V$. Each edge has a weight $d(v_i, v_j)$ corresponding to the Euclidean distance between nodes v_i, and v_j.

There are m salesmen (agents) who start their travel from the depot to visit the cities before returning to the depot. The tour of the i-th agent is defined as a sequence of nodes $T_i = (v_{i,0}, v_{i,1}, v_{i,2}, \cdots, v_{i,n_i}, v_{i,n_i+1})$, with $v_{i,0} = v_{i,n_i+1} = v_0$ being the depot. The tour will not contain any repeated nodes except for the depot, and the depot node will not appear in the middle of the tour, meaning that $\forall 1 \leq j, k \leq n_i, v_{i,j} \neq v_{i,k} \neq v_0$. We use $\{T_i\}$ to denote the set of nodes in tour $T_i, i = 1, 2, \cdots, m$, with $|T_i| = n_i + 2$ being the number of nodes in T_i. Every node except the depot must be visited exactly once by the agents, i.e., we have $\cup_{i=1}^{m}\{T_i\} = V$, and $\cap_{i=1}^{m}\{T_i\} \setminus \{v_0\} = \emptyset$. The cost of the i-th agent's tour is defined in equation (1).

$$C(T_i) = \sum_{j=1}^{|T_i|-1} d(v_{i,j-1}, v_{i,j}). \tag{1}$$

The objective of solving the m^3-TSP is to plan tours for all agents such that the maximum cost among the agents' tours is minimized, which is defined in equation (2).

$$\begin{aligned} L(z^*) &= \min \max\{C(T_i)|, 1 \leq i \leq m\} \\ s.t. &\cup_{i=1}^{m} \{T_i\} = V \\ &\cap_{i=1}^{m} \{T_i\} \setminus \{v_0\} = \emptyset \\ &\forall_{1 \leq i \leq m, 1 \leq j,k \leq n_i} T_{i,j} \neq T_{i,k} \neq v_0 \end{aligned} \tag{2}$$

where $L(z^*)$ represents the optimal cost for the m^3-TSP, and z^* is the optimal solution for m^3-TSP. In this context, z^* is defined as $T_1 + T_2 + \cdots + T_m$, where the $+$ symbol representing sequence concatenation.

3 Method

In this section, we introduce the GaS framework, as illustrated in Figure 1. We propose to first generate a path that sequentially visits all cities using an RL-based path generator and split the path to form the tours for each salesman with a splitting algorithm. The path generator employs RL to output the sequence of all cities, excluding the depot. Then we split the sequence into multiple subsequences by the splitting algorithm, which are subsequently assigned to the corresponding traveling salesmen. After the assignment is complete, the maximum cost of the tours corresponding to each subsequence is used as the reward to train the path generator.

The meta-level idea behind the design of this framework is problem decomposition. The m^3-TSP can be naturally divided into two components: a navigation component (TSP-like) and a partitioning component [35]. While solving the TSP is NP-complete [24], the partitioning part does not necessarily pose significant computational difficulty, provided an appropriate problem decomposition is adopted. For the components that are indeed hard to solve, we rely on RL, which motivates the design of the path generator. For the components that may admit polynomial-time algorithms, we design the splitting algorithm to ensure solution quality. The idea of decomposition not only enhances the interpretability and modularity of the framework, but also offers potential for integration with a broader spectrum of learning-based approaches. In the following subsections, we introduce the path generator and the splitting algorithm in detail.

3.1 The Path Generator

Given an instance of the m^3-TSP, **the path generator** is responsible for generating a path across all the cities:

$$z' = (v_{1'}, v_{2'}, \cdots, v_{i'}, \cdots, v_{n-1'}) \tag{3}$$

where $v_{i'}, 1 \leq i' \leq n-1$ is a city node defined in the graph. The path will be split and assigned to different salesmen, by the **splitting algorithm**, as detailed in the following section.

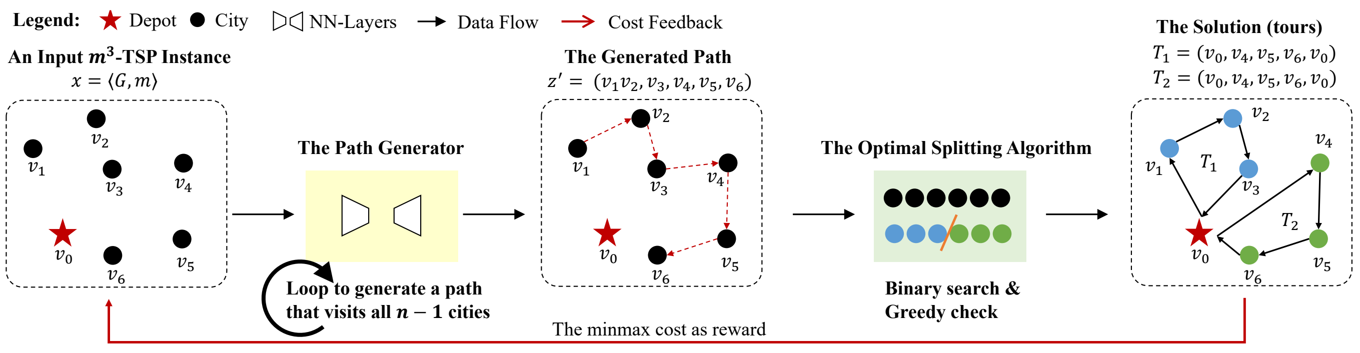
Legend: ★ Depot ● City ▷◁ NN-Layers → Data Flow → Cost Feedback
An Input m^3-TSP Instance
$x = \langle G, m \rangle$
v_0 v_1 v_2 v_3 v_4 v_5 v_6
The Path Generator
Loop to generate a path that visits all $n-1$ cities
The Generated Path
$z' = (v_1 v_2, v_3, v_4, v_5, v_6)$
The Optimal Splitting Algorithm
Binary search & Greedy check
The Solution (tours)
$T_1 = (v_0, v_4, v_5, v_6, v_0)$
$T_2 = (v_0, v_4, v_5, v_6, v_0)$
T_1 T_2
The minmax cost as reward

Figure 1. The GaS framework. The two key components are the path generator and the optimal splitting algorithm.

We formulate the problem of generating a path given an instance of m^3-TSP using the partially observable Markov Decision Processes (POMDPs) and train the path generator that generates a path step-by-step, similar to AM [17] and Eqt [27], as follows.

3.1.1 *POMDP Formulation*

Observation. We define the observation of the problem at time t as $\mathbf{s}_t = \langle \mathbf{x}_g, \mathbf{o}_t \rangle$, where $\mathbf{x}_g = \langle G, k \rangle$ contains time-independent global information, and $\mathbf{o}_t = \langle n_f, n_c \rangle$ contains observations made at time t. In $\mathbf{x}_g$, G represents the complete graph that defines an instance of the problem as introduced in the previous section, and $k = n/m$ is the ratio of the number of cities to the number of salesmen. Here, G is represented by the 2D coordinates of all cities and the depot. In $\mathbf{o}_t$, n_f represents the information of the depot, and n_c represents the information of the currently visited city.

From the definition of the observation, it is clear that key information (state) necessary for solving the problem, such as the number of traveling salesmen currently in use, is not included. This omission arises because our framework delegates the decision of when to return to the depot to the splitting algorithm, making such information unavailable during the policy execution. Unlike work such as Eqt [27], our observation definition adheres to the principle of simplicity, retaining only the essential information that can be obtained, without introducing additional heuristics. The parts that are difficult to define explicitly are instead learned by the LSTM.

Action. The action a_t can be any unvisited city at time t. Notably, the depot is not considered a city, so the action sequence will exclude the depot. After all cities have been visited, the action sequence forms z', which is the result of the path generator.

Reward. The reward is the negative cost of the maximum of $C(T_i)$ as described in equation (1) at the final time step. At all other steps, the reward is zero. To obtain a reward from z', it must first be split by the splitting algorithm described in the next section.

3.1.2 *Policy Description*

When addressing the problem using RL, we aim to learn a policy $\pi_\theta(a_t \mid \mathbf{s}_{1:t})$, which defines a probability distribution over the action space conditioned on the sequence of past observations. The objective is to maximize the expected cumulative reward, which, in our setting, corresponds to the optimization goal of the m^3-TSP. The probability distribution over a complete solution path is denoted as $p_\theta(z'|x)$, where x denotes a specific m^3-TSP instance, z' is a candidate solution, and θ represents the learnable parameters of the policy, as detailed in the next section.

3.2 *Network Architecture of the Path Generator*

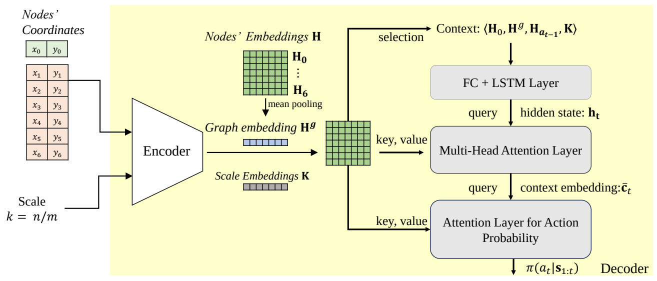
Nodes' Coordinates
x_0 y_0
x_1 y_1
x_2 y_2
x_3 y_3
x_4 y_4
x_5 y_5
x_6 y_6
Scale
$k = n/m$
Encoder
Nodes' Embeddings $\mathbf{H}$
$\mathbf{H_0}$
$\mathbf{H_6}$
mean pooling
Graph embedding $\mathbf{H}^g$
Scale Embeddings $\mathbf{K}$
selection
Context: $\langle \mathbf{H}_0, \mathbf{H}^g, \mathbf{H}_{a_{t-1}}, \mathbf{K} \rangle$
FC + LSTM Layer
query
hidden state: $\mathbf{h_t}$
key, value
Multi-Head Attention Layer
query
context embedding: $\bar{\mathbf{c}}_t$
key, value
Attention Layer for Action Probability
$\pi(a_t|\mathbf{s}_{1:t})$
Decoder

Figure 2. The network architecture of the path generator.

The splitting algorithm can only be applied after the entire path is generated, which presents a challenge for training the path generator by RL, known as the partial observation problem due to delayed splitting. From the definition of the POMDP, we observe that at each time step, the state should include information about which salesman is taking action at this time step. Methods like Eqt [27] include the "return to the depot" in the action space, allowing the environment to naturally switch to the next traveling salesman and update state transitions accordingly.

To tackle this issue, inspired by the RL for POMDPs, we introduce an LSTM-based structure in the decoder that efficiently estimates the true state under the current policy execution history for a given m^3-TSP instance. This structure constructs a hidden state based on the observation history and makes subsequent decisions based on the hidden state. The network structure is shown in Figure 2.

3.2.1 *Encoder*

The encoder takes as input the two-dimensional coordinate data of the nodes and the total number of traveling salesmen. The output includes node embedding, a graph embedding and scale embedding for the m^3-TSP instance. The initial embedding of all nodes is computed by applying a fully connected layer to transform the node coordinates, where distinct transformations are employed for the depot node and the remaining city nodes. Following this, multiple attention

Algorithm 1 Greedy Check

```
Input: $G$, graph consisting of a depot and cities;
Input: $m$, the number of agents;
Input: $z'$, path generated by the path generator;
Input: $c_m$, the threshold to check.
Output: whether the threshold can be successfully achieved.
 1: $i \leftarrow 0$
 2: initialize $T_i$ as $(v_0, )$
 3: for $t = 1, 2, \cdots, n-1$ do
 4:     if $C(T_i + z'_t + v_0) \leq c_m$ // equation (1) then
 5:         $T_i \leftarrow T_i + z'_t$
 6:     else
 7:         $T_{i+1} = (v_0, z'_t, )$
 8:         if $i + 1 \geq m$ or $C(T_{i+1} + v_0) > c_m$ then
 9:             return false
10:         end if
11:         $i \leftarrow i + 1$
12:     end if
13: end for
14: return true
```

layers as described in [17] are employed to process the node embedding, culminating in the embedding $\mathbf{H}$ for all nodes. The embedding of the i-th node is denoted by $\mathbf{H}_i$. To derive the graph embedding $\mathbf{H}^g$, we employ mean pooling [23] across the node embedding. The scale embedding $\mathbf{K}$ is acquired by mapping the scale $k = n/m$ of a m^3-TSP instance into the embedding space via a fully connected layer. We employ scale embedding to furnish the model with scale-specific information, thereby enabling it to generalize across various problem scales.

3.2.2 Decoder

The decoder takes the nodes' embedding and scale embedding as input and outputs the probability of each legal action being selected at each time step. Compared to the classical decoder structure, we have added several components to mitigate partial observation problems. The relevant concepts are presented as follows.

Context: At the t-th step of the path generation, $t-1$ actions have already been generated, denoted as $\langle a_1, a_2, \cdots, a_{t-1}\rangle$. The context of the problem at this step is defined as the concatenation of vectors: $\mathbf{c}_t = [\mathbf{H}^g; \mathbf{H}_0, \mathbf{H}a_{t-1}; \mathbf{K}]$. The vector $\mathbf{c}_t$ is then passed through a fully connected layer, the result is denoted as $\hat{\mathbf{c}}_t$.

Hidden State: To mitigate the partial observability problem, we employ an LSTM [10] to process the context vector $\hat{\mathbf{c}}_t$ and maintain a hidden state $\mathbf{h}_t$. Specifically, we adopt an incremental formulation, where the hidden state is updated as $\mathbf{h}_t = \mathrm{LSTM}(\hat{\mathbf{c}}_t, \mathbf{h}_{t-1})$, with the initial states $\mathbf{h}_0$ set to zero vectors.

A new context embedding from hidden state is computed by a multi-head attention layer $\overline{\mathbf{c}}_t = \mathbf{MHA}(\mathbf{h}_t, \mathbf{H}, \mathbf{H})$ with appropriate mask. Finally, $\overline{\mathbf{c}}_t$ is used as a contextual query, with all valid nodes' embedding serving as keys and values, to construct an attention-based policy $\pi_\theta(a_t|\mathbf{s}_{1:t})$, consistent with the AM [17].

3.3 The Splitting Algorithm

After the path is generated, we need to split it to obtain the solution for the m^3-TSP. There are various possible splitting schemes, and each results in a different cost. Our goal is to efficiently compute the cost corresponding to the optimal splitting scheme, which will be used as the reward for RL training. Assuming the path generator

Algorithm 2 The Splitting Algorithm

```
Input: $G$, graph consisting of a depot and cities;
Input: $m$, the number of agents;
Input: $z'$, path generated by the path generator.
Output: $c_m$ is the near optimal cost.
1: $L = 0, R = \hat{c}$ // binary search boundary
2: while $R - L > \epsilon$ do
3:     $c_m \leftarrow (L + R)/2$
4:     if GreedyCheck$(G, m, z')$ then
5:         $L \leftarrow c_m$
6:     else
7:         $R \leftarrow c_m$
8:     end if
9: end while
```

provides the visiting path z' for all salesmen, we need to split the path into m subsequences and insert v_0 as the first and last elements of the subsequences to form tours $\{T_i | 1 \leq i \leq m\}$ for each salesman. The min-max cost of the tours can be calculated in the splitting algorithm, as defined by equation (1). A brute-force algorithm can enumerate all possible splitting points to obtain the optimal result with a $O(n^{m-1})$ time complexity. The time complexity of such an algorithm increases exponentially with the number of salesmen, which is not suitable for fast training. To address this, we design a splitting algorithm with near-linear time complexity with respect to the number of nodes, which is outlined below.

Firstly, we convert this problem into a decision problem as follows: *Given a path z', can it be visited in order by no more than m salesmen such that the tour length of any salesmen does not exceed a predefined threshold c_m?* This decision problem can be addressed using a greedy algorithm, referred to as "greedy check" as listed in Algorithm 1. In the algorithm block, we use "+" for sequence concatenation. In this greedy check, each traveling salesman attempts to visit as many cities as possible. The answer to the original decision problem is determined by whether more than m salesmen are required by greedy check. If the greedy check uses more than m salesmen, then no other method can use no more than m salesmen to finish the task, so the answer to the decision problem is "no". Conversely, it provides a solution to the decision problem, so the answer to the decision problem is "yes". Secondly, If the decision problem is satisfied for a given c_m, then it will also be satisfied for any $c \geq c_m$. Otherwise, it cannot be satisfied for any $c \leq c_m$. This suggests that the minimal c_m that satisfies the decision problem can be efficiently determined using binary search. By synthesizing the aforementioned aspects, we introduce the splitting algorithm detailed in Algorithm 2. The time complexity of the algorithm is $O(n \log \frac{\hat{c}}{\epsilon})$, and the approximation ratio is $1 + \epsilon$ (or $1 + \epsilon/c^*$ if we assume $c^* < 1$, where c^* is the optimal cost). The cost reported by the splitting algorithm is used as a reward for RL, denoted as $L(z')$.

As mentioned earlier, if the greedy check can correctly answer the decision problem, binary search can find the optimal solution within the error margin. We provide its proof in the Appendix [29].

3.4 The Training Process

RL trains through the REINFORCE [31], with the gradient of the objective function given by Equation (4).

$$\nabla_\theta = \mathrm{E}_{p_\theta(z'|x)}[\nabla \log p_\theta(z'|x)(L(z') - b(x))] \tag{4}$$

In Equations (4), x represents an instance of the m^3-TSP, $p_\theta(z'|x)$

denotes the probability of the policy generating the path z' condition on x, and $b(x)$ denotes the baseline, following [27], which is computed as the average reward over multiple trajectories generated by applying data augmentation to the input instance x. Several similar symmetry-based data augmentation techniques can also be found in [33, 32, 34]. Finally, a gradient descent-based method that can be used to optimize the problem.

4 Experimental Results

In this section, we aim to answer the following research questions:

1. RQ1 [Solution Quality]. Is the proposed GaS framework effective and efficient in finding high-quality solutions?
2. RQ2 [Transferability]. Is a learned solver still effective when we apply it to problems different from the training scenarios in terms of scales and city distributions?

4.1 Baseline Selection

Several deep RL-based methods have been proposed for the m^3-TSP. Among them, we select two of the most recent and competitive algorithms for comparison: Eqt [27] and Dpn [35]. Our selection is based on the following criteria: (1) both methods have demonstrated impressive empirical performance and, to the best of our knowledge, represent the most recent and state-of-the-art learning-based approaches; and (2) both are fully open-sourced, enabling fair comparisons under identical experimental settings and ensuring the reproducibility of results. We directly used the publicly available models of Eqt and Dpn for evaluation. Since Eqt does not provide a pre-trained model for instances with 100 nodes, we trained Eqt for this setting using its publicly released training script.

For classical heuristic methods, following the evaluation protocol established in Eqt [27], we report the performance of LKH3 [9] on the m^3-TSP. This allows us to quantify the performance gap between learning-based approaches and traditional heuristic algorithms. HGA [21] is another competitive heuristic solver, but due to space constraints, we restrict our comparisons to LKH3. We report heuristic solver results under 60 and 600 second time limits.

4.2 Experiment Design

We conduct experiments under the following controlled settings.

To address **RQ1**, we randomly generate 2D coordinates for the cities and depot, uniformly distributed within a 1×1 rectangular area during training. The RL-based policy is trained under two settings: $n = 50$ and $n = 100$ nodes. The number of salesmen is randomly sampled from a uniform distribution in the range $[2, 10]$. We then validate the trained GaS framework on 100 randomly generated instances, using the same test data as in Eqt [27] and Dpn [35]. These instances are not part of the training process.

The answer to **RQ1** focuses on the solution quality of the GaS framework for problems of the same scale and distribution. However, in practical applications, we often encounter large-scale problems [19] and out-of-distribution data [37]. Instances of varying scales are also crucial. To assess the broader applicability of the framework, we conduct additional experiments to address **RQ2**. Specifically, we perform three types of experiments as follows:

1. **RQ2.1**: How does the trained model perform on data with different distributions? To answer this, we directly apply the learned model to three out-of-distribution datasets proposed in [37] and compare the results with baseline methods.
2. **RQ2.2**: How does the trained model perform on large-scale problems? To answer this, we employ a similar fine-tuning process on scenarios with 200 and 500 nodes, as described in [35], and compare the results with baseline methods on large-scale problems.
3. **RQ2.3**: How does the trained model perform when the scale differs from the training settings? To answer this, we evaluate the fine-tuned model on $n = 200$ and $n = 500$ settings with varying numbers of salesmen.

Our code and data are publicly available [29]. For further details regarding reproducibility, please refer to Appendix [29].

4.3 Solution Quality Results (RQ1)

The experimental results for RQ1 are summarized in Table 1. In the table, $\times k$. represents the generation of k samples and reporting the best result. For Eqt and GaS, $\times 8$ indicates 8-fold data augmentation [18], while $\times 128$ refers to performing 16 sampling width [17] for each instance after 8-fold data augmentation, and reporting the best result. On the other hand, Dpn utilizes the data augmentation and agent permutation techniques described in the referenced paper. As shown in Table 1, the GaS method achieves better results than the best existing learning-based methods in 13 out of 18 experimental configurations, and matches their performance in one other. One-sided Wilcoxon tests [30] show that GaS significantly reduces the cost compared to Dpn and Eqt, with p-values of 9.66×10^{-29} / 1.10×10^{-42} (50 nodes) and 2.91×10^{-7} / 4.00×10^{-80} (100 nodes). A comparison between learning-based methods and LKH3 reveals that existing learning-based approaches still lag behind search-based methods like LKH3 in terms of solution quality, particularly when the value of m is small. This indicates that when the problem characteristics are closer to the classical TSP, current learning-based methods tend to perform poorly and leave greater room for improvement. In contrast, when m is large, learning-based methods are already capable of solving the problem effectively, leaving less room for further optimization. We do not report solution time in the table because all learning-based approaches solve each instance in under one second at this scale, making the differences among them negligible. Nevertheless, compared to the search-based LKH3, learning-based methods demonstrate a clear advantage in terms of inference efficiency.

4.4 Transferability Results (RQ2)

Results for RQ2.1: The trained model should be applicable to data with distributions different from the training distribution. We evaluated the model on datasets with Rotation, Gaussian, and Explosion distributions [37]. A sample of the data is shown in Figure 3, and the comparison results with Eqt and Dpn are presented in Table 2. The original dataset contains 200 nodes and 2000 instances for each distribution. For evaluation, we used the first 100 instances and the first n nodes. According to the experimental results, GaS outperforms existing learning-based methods in 41 out of 54 test configurations and achieves identical performance (to four decimal places) in 8 more. This means that in 90.7% of the test cases, GaS performs at least as well as the best current learning-based approaches.

Results for RQ2.2 & RQ2.3: Following the fine-tuning setup in Dpn, we fine-tune our model on instances with 200 nodes and 10–20 agents, as well as 500 nodes and 30–50 agents. Evaluation is then conducted on scenarios with 200 nodes and 2–20 agents, and 500

Table 1. Evaluation Results for RQ1. The average min-max costs are reported in this table, with results averaged over 100 instances. A gray background highlights the best performance among learning-based methods, while bold font indicates the best performance across all methods.

				mTSP Instances, n=50					
m	2	3	4	5	6	7	8	9	10
LKH3 (60s)	3.1520	2.4353	2.1546	2.0340	1.9807	1.9515	1.9481	1.9400	1.9431
LKH3 (600s)	**3.1517**	**2.4338**	**2.1502**	**2.0234**	**1.9711**	**1.9440**	1.9358	1.9336	1.9317
Eqt ×1	3.3262	2.5643	2.2402	2.0798	1.9976	1.9618	1.9414	1.9349	1.9321
Eqt ×8	3.2343	2.4901	2.1870	2.0399	1.9788	1.9465	1.9359	1.9324	1.9303
Eqt ×128	3.2079	2.4700	2.1732	2.0315	1.9742	1.9443	**1.9349**	**1.9321**	1.9303
Dpn ×1	3.2576	2.5149	2.2038	2.0545	1.9979	1.9549	1.9430	1.9352	1.9334
Dpn ×8	3.1956	2.4654	2.1735	2.0337	1.9744	1.9460	1.9362	1.9323	1.9304
Dpn ×128	3.1949	2.4637	2.1728	2.0323	1.9736	1.9454	1.9358	1.9323	**1.9302**
GaS ×1	3.3339	2.5448	2.2216	2.0575	1.9923	1.9572	1.9396	1.9338	1.9312
GaS ×8	3.2108	2.4723	2.1788	2.0367	1.9771	1.9451	1.9354	1.9323	1.9307
GaS ×128	3.1918	2.4598	2.1681	2.0301	1.9739	1.9440	**1.9349**	1.9322	1.9305
				mTSP Instances, n=100					
m	2	3	4	5	6	7	8	9	10
LKH3 (60s)	4.0713	2.9551	2.4814	2.2397	2.1119	2.0580	2.0360	2.0225	2.0090
LKH3 (600s)	**4.0694**	**2.9436**	**2.4572**	**2.2058**	**2.0719**	**2.0076**	1.9799	1.9689	1.9640
Eqt ×1	4.5123	3.2110	2.6303	2.3239	2.1528	2.0557	2.0036	1.9760	1.9634
Eqt ×8	4.3148	3.1044	2.5549	2.2646	2.1083	2.0290	1.9845	1.9637	1.9549
Eqt ×128	4.2524	3.0672	2.5280	2.2469	2.0947	2.0164	1.9792	1.9611	1.9531
Dpn ×1	4.2705	3.0801	2.5498	2.2704	2.1177	2.0335	1.9921	1.9724	1.9587
Dpn ×8	4.1804	3.0233	2.5041	2.2346	2.0909	2.0143	1.9804	1.9623	1.9534
Dpn ×128	4.1774	3.0181	2.5015	2.2315	2.0895	2.0128	1.9793	1.9613	1.9531
GaS ×1	4.3156	3.1230	2.5766	2.2796	2.1240	2.0429	1.9996	1.9739	1.9584
GaS ×8	4.2267	3.0439	2.5114	2.2417	2.0964	2.0190	1.9807	1.9625	1.9539
GaS ×128	4.1852	3.0157	2.4974	2.2285	2.0859	2.0120	**1.9764**	**1.9596**	**1.9524**

Table 2. Evaluation Results for RQ2.1. The average min-max cost are reported in this table. All results are averaged over 100 out of distribution instances. A gray background indicates the best performance among learning-based methods.

m	2	3	4	5	6	7	8	9	10
				Rotation, n = 50					
Eqt ×128	2.5622	2.0049	1.7752	1.6654	1.6171	1.5971	1.5879	1.5844	1.5827
Dpn ×128	2.5138	1.9650	1.7499	1.6501	1.6102	1.5917	1.5851	1.5826	1.5822
GaS ×128	2.5137	1.9564	1.7450	1.6471	1.6085	1.5905	1.5848	1.5826	1.5821
				Gaussian, n = 50					
Eqt ×128	2.1551	1.6405	1.4487	1.3725	1.3464	1.3368	1.3326	1.3318	1.3312
Dpn ×128	2.0672	1.5964	1.4250	1.3612	1.3406	1.3340	1.3318	1.3311	1.3310
GaS ×128	2.0535	1.5897	1.4216	1.3597	1.3404	1.3338	1.3318	1.3311	1.3310
				Explosion, n = 50					
Eqt ×128	2.5678	2.0384	1.8003	1.7099	1.6628	1.6449	1.6366	1.6351	1.6344
Dpn ×128	2.5231	2.0057	1.7867	1.6970	1.6538	1.6402	1.6355	1.6343	1.6341
GaS ×128	2.5177	2.0023	1.7846	1.6962	1.6550	1.6400	1.6354	1.6342	1.6341
				Rotation, n = 100					
Eqt ×128	3.5046	2.5504	2.1087	1.8745	1.7562	1.6927	1.6635	1.6468	1.6392
Dpn ×128	3.3594	2.4615	2.0548	1.8440	1.7411	1.6858	1.6571	1.6439	1.6382
GaS ×128	3.3112	2.4328	2.0340	1.8378	1.7365	1.6854	1.6551	1.6432	1.6369
				Gaussian, n = 100					
Eqt ×128	3.1191	2.2778	1.8371	1.6062	1.4987	1.4519	1.4315	1.4243	1.4230
Dpn ×128	2.9806	2.1438	1.7532	1.5704	1.4856	1.4478	1.4296	1.4235	1.4223
GaS ×128	2.9184	2.0929	1.7326	1.5657	1.4833	1.4474	1.4309	1.4240	1.4225
				Explosion, n = 100					
Eqt ×128	3.4369	2.5637	2.1442	1.9181	1.7997	1.7446	1.7180	1.7061	1.7007
Dpn ×128	3.2976	2.4687	2.0818	1.8938	1.7908	1.7393	1.7153	1.7034	1.6991
GaS ×128	3.2806	2.4472	2.0756	1.8908	1.7898	1.7393	1.7154	1.7043	1.6991

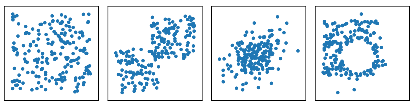

Figure 3. City locations under different distributions. From left to right: Uniform, Rotation, Gaussian, Explosion.

nodes and 10–40 agents. The results appear in Figure 4. As shown in the figure, the min-max cost remains unchanged once the number of agents exceeds a certain threshold. For the 200-node and 500-node scenarios, GaS reaches this threshold at $m = 16$ and $m = 33$, with corresponding costs of 1.9628 and 2.0061, respectively. Beyond these points, adding more agents yields no further improvement. This also suggests that the number of agents used in existing evaluations for larger-scale scenarios may be suboptimal, as it often falls within a saturated range, where the optimal solution is effectively bounded by twice the distance between the depot and the farthest city node, thus limiting the potential difficulty of the instance. For larger-scale problems (e.g., $n = 1000, m \geq 50$; $n = 2000, m \geq 100$; and $n = 5000, m \geq 300$)—all of which are configurations used in Eqt for performance evaluation—we empirically verify that the model fine-tuned on the $n = 500$ setting consistently yields optimal solutions. Therefore, we do not include additional comparisons for these configurations. Consistent with the findings in RQ1, the more challenging aspects for current learning-based frameworks tend to emerge when the problem characteristics become more similar to those of the classical TSP. From the portion of the figure where the number of agents decreases, we observe that when a scale shift occurs, GaS demonstrates better transferability.

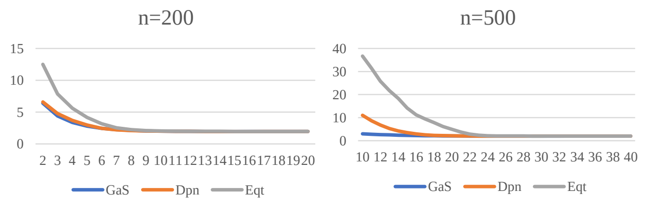

Figure 4. Performance of Finetuned Models on Larger Problem Instances. X-axis: number of agents; Y-axis: min-max cost.

4.5 Ablation study

We conduct an ablation study to assess the contribution of each component in the GaS framework. The results are summarized in Table 3. Specifically, we examine the effects of removing the LSTM-based decoder and the optimal splitting component, denoted as w/o LSTM and w/o optimal splitting, respectively. Without the LSTM, the context embedding is processed by a multilayer perceptron (MLP) and passed directly to the multi-head attention layer. When the optimal splitting module is removed, the action of returning to the depot is no longer masked, and the decision of when to return is entirely governed by the RL policy. The results demonstrate that incorporating the LSTM effectively mitigates partial observability and enhances solution quality. Moreover, the inclusion of optimal splitting allows the RL policy to concentrate on navigation-related decisions rather than the splitting strategy, leading to substantial performance improvements, particularly in scenarios with a small number of agents.

Table 3. Ablation study on the lstm-based decoder and optimal splitting.

mTSP, n=50				
m=	2	3	5	7
w/o LSTM	3.2272	2.4862	2.0489	1.9482
w/o optimal splitting	4.3563	3.2291	2.0608	1.9470
GaS $\times 128$	**3.1918**	**2.4598**	**2.0301**	**1.9440**

5 Related Work

Combinatorial optimization problems are a category of problems that focus on optimization within discrete spaces [22]. One of the most well-known combinatorial optimization problems is the Traveling Salesman Problem [24] (TSP), with many learning-based solvers [26, 14, 11]. As an important generalization, the m^3-TSP introduces additional complexity by involving multiple salesmen. We summarize existing RL-based approaches for solving the m^3-TSP as follows. NCE [16] learns to iteratively refine an initial solution through cross-exchange operations. Although this method can achieve high-quality solutions given sufficient time, it incurs a relatively high computational cost. DisPN [12] develops an offline policy for assigning cities to salesmen using graph networks and reinforcement learning, while relying on traditional solvers to construct tours for each salesman. DAN [2] employs a decentralized attention-based model to learn collaborative policies for online tour construction. ScheduleNet [25] leverages type-aware graph attention to learn assignments between salesmen and cities, enabling multiple salesmen to plan tours concurrently in an online setting. SplitNet [20], which is most similar to our approach, uses classical heuristics to generate a city sequence and learns a splitting strategy to divide it into individual tours. In contrast, we adopt an optimal splitting algorithm to further improve performance. AMARL [7] introduces a gated transformer-based state representation to coordinate multiple agents in solving the m^3-TSP via RL. However, the multi-agent setting increases training complexity, while our method uses a fully serial path generation strategy, simplifying learning. Equity-Transformer (Eqt) [27] sequentially generates tours for all salesmen. Although it also uses a sequential approach, it requires learning a “return-to-depot” action, which we avoid by decoupling path generation and partitioning, thereby reducing learning difficulty. Dpn [35] proposes an end-to-end model that jointly learns navigation and partition. In contrast, our method focuses solely on learning the navigation component, leaving the partitioning to be handled optimally by the optimal splitting algorithm. UDC [36] also follows a similar divide-and-conquer paradigm for solving combinatorial problems. While we adopt an algorithmic framework augmented with learning, they employ a fully learning-based framework.

6 Conclusion

We introduce the Generate and Split (GaS) framework for solving the m^3-TSP. This framework comprises an LSTM-based model combined with RL for the path generator and a deterministic, approximately optimal splitting algorithm. To train the path generator alongside the optimal splitting algorithm, we propose an LSTM-enhanced decoder that takes into account the observation history to mitigate the partial observability issue caused by delayed splitting. Experimental results demonstrate that the GaS framework outperforms existing learning-based methods on both solution quality and transferability.

Limitations and future work: Unlike fully learning-based methods, the GaS framework combines algorithmic design with reinforcement learning, which may limit its immediate generality. Nevertheless, combining algorithmic priors with end-to-end learning provides a promising approach, and we plan to apply it to other combinatorial optimization problems in our future work.

Acknowledgements

This work is supported by the National Key Research and Development Project of China No. 2023YFC3107100, NSFC No. 62172203, the Collaborative Innovation Center of Novel Software Technology and Industrialization.

References

[1] A. Bogyrbayeva, T. Yoon, H. Ko, S. Lim, H. Yun, and C. Kwon. A deep reinforcement learning approach for solving the traveling salesman problem with drone. *Transportation Research Part C: Emerging Technologies*, 148:103981, 2023.

[2] Y. Cao, Z. Sun, and G. Sartoretti. Dan: Decentralized attention-based neural network to solve the minmax multiple traveling salesman problem. *arXiv preprint arXiv:2109.04205*, 2021.

[3] H. Chakraa, E. Leclercq, F. Guérin, and D. Lefebvre. A multi-robot mission planner by means of beam search approach and 2-opt local search. In *2023 9th International Conference on Control, Decision and Information Technologies (CoDIT)*, pages 1–6. IEEE, 2023.

[4] I. I. Cplex. V12. 1: User's manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

[5] P. M. França, M. Gendreau, G. Laporte, and F. M. Müller. The m-traveling salesman problem with minmax objective. *Transportation Science*, 29(3):267–275, 1995.

[6] F. Gan and D. Liu. Forest fire fast response method based on optimized uav algorithm. In *2022 7th International Conference on Intelligent Computing and Signal Processing (ICSP)*, pages 47–51. IEEE, 2022.

[7] H. Gao, X. Zhou, X. Xu, Y. Lan, and Y. Xiao. Amarl: An attention-based multiagent reinforcement learning approach to the min-max multiple traveling salesmen problem. *IEEE Transactions on Neural Networks and Learning Systems*, 2023.

[8] M. J. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. In *AAAI fall symposia*, volume 45, page 141, 2015.

[9] K. Helsgaun. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*, 12:966–980, 2017.

[10] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[11] L. Holbein and Y. Schmid. Reinforcement learning of tsp heuristics with message passing neural networks. 2023.

[12] Y. Hu, Y. Yao, and W. S. Lee. A reinforcement learning approach for optimizing multiple traveling salesman problems over graphs. *Knowledge-Based Systems*, 204:106244, 2020.

[13] Z. Iklassov, I. Sobirov, R. Solozabal, and M. Takáč. Reinforcement learning for solving stochastic vehicle routing problem. In *Asian Conference on Machine Learning*, pages 502–517. PMLR, 2024.

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

[15] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[16] M. Kim, J. Park, and J. Park. Learning to cross exchange to solve min-max vehicle routing problems. In *The Eleventh International Conference on Learning Representations*, 2023.

[17] W. Kool, H. van Hoof, and M. Welling. Attention, learn to solve routing problems!, 2019. URL https://arxiv.org/abs/1803.08475.

[18] Y.-D. Kwon, J. Choo, B. Kim, I. Yoon, Y. Gwon, and S. Min. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems*, 33:21188–21198, 2020.

[19] S. Li, Z. Yan, and C. Wu. Learning to delegate for large-scale vehicle routing. *Advances in Neural Information Processing Systems*, 34: 26198–26211, 2021.

[20] H. Liang, Y. Ma, Z. Cao, T. Liu, F. Ni, Z. Li, and J. Hao. Splitnet: a reinforcement learning based sequence splitting method for the minmax multiple travelling salesman problem. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 8720–8727, 2023.

[21] S. Mahmoudinazlou and C. Kwon. A hybrid genetic algorithm for the min–max multiple traveling salesman problem. *Computers & Operations Research*, 162:106455, 2024.

[22] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134:105400, 2021.

[23] D. Mesquita, A. Souza, and S. Kaski. Rethinking pooling in graph neural networks. *Advances in Neural Information Processing Systems*, 33: 2220–2231, 2020.

[24] C. H. Papadimitriou. The euclidean travelling salesman problem is np-complete. *Theoretical computer science*, 4(3):237–244, 1977.

[25] J. Park, C. Kwon, and J. Park. Learn to solve the min-max multiple traveling salesmen problem with reinforcement learning. In *AAMAS*, volume 22, pages 878–886, 2023.

[26] M. V. Seiler, J. Rook, J. Heins, O. L. Preuß, J. Bossek, and H. Trautmann. Using reinforcement learning for per-instance algorithm configuration on the tsp. In *2023 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 361–368. IEEE, 2023.

[27] J. Son, M. Kim, S. Choi, H. Kim, and J. Park. Equity-transformer: Solving np-hard min-max routing problems as sequential generation with equity context. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 20265–20273, 2024.

[28] R. Sun, C. Tang, J. Zheng, Y. Zhou, and S. Yu. Multi-robot path planning for complete coverage with genetic algorithms. In *Intelligent Robotics and Applications: 12th International Conference, ICIRA 2019, Shenyang, China, August 8–11, 2019, Proceedings, Part V 12*, pages 349–361. Springer, 2019.

[29] W. Wang. Generate-and-split (gas) framework for m^3-tsp: Code and data. https://github.com/wenwenla/ecai-2025-mtsp-solver.git, 2025. Accessed: 2025-08-23.

[30] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945. doi: 10.2307/3001968.

[31] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[32] X. Yu, R. Shi, P. Feng, Y. Tian, J. Luo, and W. Wu. Esp: Exploiting symmetry prior for multi-agent reinforcement learning. In *ECAI 2023*, pages 2946–2953. IOS Press, 2023.

[33] X. Yu, R. Shi, P. Feng, Y. Tian, S. Li, S. Liao, and W. Wu. Leveraging partial symmetry for multi-agent reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17583–17590, 2024.

[34] X. Yu, Y. Tian, L. Wang, P. Feng, W. Wu, and R. Shi. Adaptaug: Adaptive data augmentation framework for multi-agent reinforcement learning. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10814–10820. IEEE, 2024.

[35] Z. Zheng, S. Yao, Z. Wang, X. Tong, M. Yuan, and K. Tang. Dpn: decoupling partition and navigation for neural solvers of min-max vehicle routing problems. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[36] Z. Zheng, C. Zhou, T. Xialiang, M. Yuan, and Z. Wang. Udc: A unified neural divide-and-conquer framework for large-scale combinatorial optimization problems. *Advances in Neural Information Processing Systems*, 37:6081–6125, 2024.

[37] J. Zhou, Y. Wu, W. Song, Z. Cao, and J. Zhang. Towards omni-generalizable neural methods for vehicle routing problems. In *International Conference on Machine Learning*, pages 42769–42789. PMLR, 2023.

A Technical Appendix

A.1 Proof of Correctness for the Greedy Check

Proof. Suppose $s = (1, p_1, p_2, \cdots, p_i, \cdots, p_{m-1}, n)$ are the split points, $p_m = n$, then the i-th subsequence t_i is defined as

$$(z'_{s_i}, \cdots, z'_j, \cdots, z'_{s_{i+1}-1}), \forall j, s_i < j < s_{i+1} - 1. \tag{5}$$

The tour for the i-th salesman is defined as T_i, by adding depot nodes to both ends of t_i as follows:

$$(v_0, z'_{s_i}, \cdots, z'_j, \cdots, z'_{s_{i+1}-1}, v_0). \tag{6}$$

The cost of T_i is defined as follows, where d represents the Euclidean distance metric.

$$C(T_i) = \sum_{j=1}^{|T_i|-1} d(T_{i,j}, T_{i,j+1}). \tag{7}$$

Since d is a distance metric, the following inequality holds:

$$d(i,j) + d(j,k) \geq d(i,k), \forall i, j, k \tag{8}$$

Assume the greedy check outputs the following split points: $s = (1, p_1, p_2, \cdots, p_{m-1}, n)$. If the split points meet the requirements, then the "greedy check" finds a solution, and outputs true.

Otherwise, we need to prove that no split points exist that can meet the requirements.

Due to the failure of the greedy check, we can draw the following conclusions, in which $|s$ means condition on split points s.

$$C(T_m|s) > c_m, \tag{9}$$

$$C(T_i|s) \leq c_m, \forall 1 \leq i < m. \tag{10}$$

Assume there exists a valid splitting point, denoted as $\hat{s} = (1, \hat{p_1}, \hat{p}_2, \cdots, \hat{p}_{m-1}, n)$, and the following inequality holds.

$$C(T_i|\hat{s}) \leq c_m, \forall 1 \leq i \leq m. \tag{11}$$

Stage 1 According to equations (9) and (11), the following equation holds:

$$\hat{p}_{m-1} > p_{m-1} \tag{12}$$

The proof is as follows:

$$\begin{aligned}
&C(T_m|s) > c_m, C(T_m|\hat{s}) \leq c_m \\
\Rightarrow &C(T_m|s) > C(T_m|\hat{s}) \\
\Rightarrow &d(v_0, z'_{p_{m-1}}) + \left(\sum_{k=p_{m-1}}^{n-2} d(z'_k, z'_{k+1})\right) + d(z'_{n-1}, v_0) > \\
&d(v_0, z'_{\hat{p}_{m-1}}) + \left(\sum_{k=\hat{p}_{m-1}}^{n-2} d(z'_k, z'_{k+1})\right) + d(z'_{n-1}, v_0)
\end{aligned} \tag{13}$$

From equation (13), we can observe $p_{m-1} \neq \hat{p}_{m-1}$. if $\hat{p}_{m-1} < p_{m-1}$, rewrite equation (13) as follows:

$$d(v_0, z'_{p_{m-1}}) > d(v_0, z'_{\hat{p}_{m-1}}) + \sum_{k=\hat{p}_{m-1}}^{p_{m-1}-1} d(z'_k, z'_{k+1}). \tag{14}$$

Equations (14) and (8) are contradictory, therefore equation (12) must hold.

Stage 2 Next, we use mathematical induction to prove $\forall 1 \leq i \leq m-1, p_i \geq \hat{p}_i$.

First, $p_1 \geq \hat{p}_1$. This is guaranteed by the "greedy check" and equation (8).

Then, assume $p_k \geq \hat{p}_k$. Next, we will prove that $p_{k+1} \geq \hat{p}_{k+1}$.

There are two possible cases, and we discuss each separately.

1. If $p_k \geq \hat{p}_{k+1}$, then $p_{k+1} \geq \hat{p}_{k+1}$.
2. If $\hat{p}_k \leq p_k < \hat{p}_{k+1}$.

Let's use some simplified notation, $0 \to p_i \sim p_{i+1} \to 0$ means:

$$d(v_0, z'_{p_i}) + \left(\sum_{j=p_i}^{p_{i+1}-2} d(z'_j, z'_{j+1})\right) + d(z'_{p_{i+1}-1}, v_0) \tag{15}$$

From $p_k \geq \hat{p}_k$ and equation (8), we can derive the following conclusion:

$$0 \to p_k \sim \hat{p}_{k+1} \to 0 \leq 0 \to \hat{p}_k \sim \hat{p}_{k+1} \to 0 \leq c_m \tag{16}$$

Therefore, according to equation (16), and due to the greedy nature, the conclusion $p_{k+1} \geq \hat{p}_{k+1}$ must hold.

Therefore, the following equation (17) must hold.

$$p_{m-1} \geq \hat{p}_{m-1} \tag{17}$$

Equations (12) and (17) are contradictory, therefore $\hat{s}$ does not exist. Therefore, the "greedy check" correctly addresses the decision problem. □

A.2 Details of the path generator

We recommend referring to our implementation for the exact code[1]. Here, we briefly describe the architecture.

The encoder takes as input a graph represented by the 2D coordinates of all nodes with shape $(\text{batch\_size}, n_{\text{nodes}}, 2)$. It produces both node-level and graph-level embeddings of dimension 128. The architecture is organized as follows:

- **Input embedding.** Separate linear layers are used to initialize three types of embeddings: (i) customer nodes, (ii) the depot node, and (iii) a scale feature defined as the ratio between the number of nodes and the number of agents.
- **Graph encoder.** The initialized embeddings are concatenated and passed through three stacked Graph Attention (GAT) layers, which capture spatial and structural dependencies between nodes.
- **Output.** The encoder outputs (i) node embeddings of size $(\text{batch\_size}, n_{\text{nodes}}, 128)$, (ii) a graph embedding obtained by mean pooling over all nodes, and (iii) the scale embedding.

The decoder is responsible for sequentially generating node selections based on the encoder outputs and contextual information. Its design integrates recurrent modeling with attention, and can be summarized as follows:

- **Context embedding.** At each decoding step, the context vector is constructed by concatenating the graph embedding, the embedding of the first node, the embedding of the current node, and the scale embedding. This concatenated vector is projected into a 128-dimensional representation via a linear layer.

[1] https://github.com/wenwenla/ecai-2025-mtsp-solver/blob/main/net/tsp_lstm.py

- **Sequential modeling.** The projected context is passed through an LSTM with hidden size 128 to capture temporal dependencies across decoding steps. Hidden and cell states are maintained across steps, with periodic detachment to stabilize training.
- **Multi-head attention.** The context embedding is transformed into query vectors, while cached node embeddings serve as keys and values. An 8-head scaled dot-product attention mechanism is applied with masking to ensure feasibility of node selection.
- **Projection and logits.** The attended context is projected back to a 128-dimensional vector and compared with transformed node embeddings to produce logits for all nodes. Logits are scaled with a hyperparameter α using the $\tanh$ function to control their range, and infeasible nodes are masked by assigning $-\infty$.
- **Output.** The decoder outputs a compatibility score vector over nodes at each step, which defines the probability distribution for selecting the next node.

A.3 Training details

The following table 4 summarizes the hyperparameters used in training our model. During training, we employ four Nvidia V100 GPUs and complete the process on 100-node instances in under three days. Fine-tuning is completed within 12 hours.

Table 4. Training hyperparameters

Hyperparameter	**Value**
Training Epochs	100
Number of agents	2–10
Augmentation	16
Batch size	512
Random seed	1234
Optimizer	Adam
Training learning rate	1×10^{-4}
Fine-tuning learning rate	1×10^{-5}