Step-wise Deep Learning Models for Solving Routing Problems

Liang Xin[1], Wen Song[2*], Zhiguang Cao[3], Jie Zhang[1]

***Abstract*—Routing problems are very important in Intelligent Transportation Systems. Recently, a number of deep learning based methods are proposed to automatically learn construction heuristics for solving routing problems. However, these methods do not completely follow Bellman's Principle of Optimality since the visited nodes during construction are still included in the following sub-tasks, resulting in sub-optimal policies. We propose a novel step-wise scheme which explicitly removes the visited nodes in each node selection step. We apply this scheme to two representative deep models for routing problems, Pointer Network (PtrNet) and Transformer Attention Model (TAM), and significantly improve the performance of the original models. To reduce computational complexity, we further propose the approximate step-wise TAM model by modifying one layer of attention. It enables training on larger instances compared to step-wise TAM, and outperforms state-of-the-art deep models with greedy decoding strategy.**

***Index Terms*—Intelligent Transportation System, Routing Problems, Deep Learning, Deep Reinforcement Learning.**

I. Introduction

Intelligent Transportation System (ITS) has received great attention recently. By combining advanced information technology such as Internet of Things [1], Cyber-Physical systems [2] and Deep Learning [3], ITS offers a promising way to improve the safety, efficiency, and sustainability of modern transportation systems [4]. As one of the most fundamental problems in ITS, *routing problems* aim at planning the paths for one or multiple vehicles so that certain objective (e.g. total travel distance) is optimized. Travelling Salesman Problem (TSP) and Capacitated Vehicle Routing Problem (CVRP) are two typical routing problems, and are of great importance to many real-world applications of ITS [5]. However, routing problems are generally NP-hard combinatorial optimization problems [6]. While exact methods such as branch and bound [7] offer nice theoretical guarantees on the optimality, they suffer from the scalability issue due to their exponential (worst-case) complexity. In contrast, heuristic algorithms often yield sub-optimal solutions with much shorter computational time, hence are more preferred for solving large-scale routing problems in reality. However, traditional heuristic approaches rely on hand-crafted decision rules to guide the solving process, which require substantial domain knowledge and engineering effort to design, and may lead to poor solutions [8].

Recently, researchers have been focusing on using deep learning methods to automatically learn the heuristics for solving routing problems in an end-to-end fashion, without the need of using domain knowledge to design hand-crafted rules [8]–[12]. Most of these methods follow the encoder-decoder paradigm, where the encoder maps the information of locations in the routing problems (e.g. coordinates, demands) into feature embeddings, and the decoder is responsible for incrementally creating the solutions. Among these methods, Pointer Network (PtrNet) [9] and Transformer Attention Model (TAM) [12] are two representative methods. PtrNet uses Long Short-Term Memory (LSTM) [13] as encoder and decoder, and is the first modern deep learning model for combinatorial optimization problems. Motivated by the recent success of Transformer [14] in Natural Language Processing, TAM adopts the self-attention based encoder and decoder, and outperforms a wide range of (traditional or learning-based) algorithms on a series of routing problems.

Essentially, the above methods focus on learning construction heuristics, which extend an empty solution by repeatedly adding locations (or nodes) into the current partial routes until completion. This process can be viewed as a sequence of node adding decisions [11], since there is no backtracking procedure to modify previous decisions. Therefore, it should follow the Bellman's Principle of Optimality [15] that the optimal policy should only consist of optimal sub-policies. Taking TSP as an example, the salesman sequentially visits cities without repetitions by starting from one city and returning to the same city. The whole problem can be divided into a sequence of sub-problems in each of which one city is to be visited. Moreover, given the starting city and the location of the salesman, the visited cities are no longer relevant to the future sub-problems. Inspired by the Bellman's Principle of Optimality, we do not need and should not incorporate any information about the visited cities when solving a (future) sub-problem of TSP. Similarly, these properties also apply to CVRP.

Though existing deep models such as PtrNet and TAM perform well in solving routing problems, they do not completely follow the Bellman's Principle of Optimality. This is because in each decoding step, i.e. for each sub-problem, the same node embeddings are used and the irrelevant nodes are not removed. In other words, they are not solving the correct sub-problems therefore may result in sub-optimal policies. To address this issue, in this paper, we propose *step-wise* deep learning models to solve routing problems. The core idea is that, at each

* Corresponding Author.

[1]Liang Xin and Jie Zhang are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore (emails: XINL0003@e.ntu.edu.sg, zhangj@ntu.edu.sg).

[2]Wen Song is with the Institute of Marine Science and Technology, Shandong University, China (email: wensong@email.sdu.edu.cn).

[3]Zhiguang Cao is with the Department of Industrial Systems Engineering and Management, National University of Singapore, Singapore (email: zhiguangcao@outlook.com).

decoding step, we explicitly eliminate the visited and irrelevant nodes in the graph, and extract feature embeddings from the same encoder with different graphs obtained after the elimination operations. We apply this scheme to PtrNet and TAM to demonstrate its effectiveness. Specifically, we propose the step-wise PtrNet (SW-PtrNet) with supervised learning for TSP, and the step-wise TAM (SW-TAM) with reinforcement learning for both TSP and CVRP. However, the step-wise operation is computationally expensive, especially for SW-TAM, since it performs complete re-embeddings at each decoding step. To make it more efficient, we propose Approximate SW-TAM (ASW-TAM) with reasonable computational overhead, where the top attention layer in the encoder serves as part of the decoder and learns features directly for solving the correct sub-problems with irrelevant nodes removed.

We perform extensive experiments to evaluate our methods. Results show that both SW-PtrNet and SW-TAM significantly outperform their respective original models on small problems. Moreover, ASW-TAM enables training on larger instances, and outperforms state-of-the-art deep models with greedy decoding strategy. Note that our aim here is not to compete with highly optimized and specialized solvers (e.g. Concorde for TSP) on a specific problem. Instead, we propose an effective scheme that can improve existing deep models towards automatically learning stronger heuristics, which could exploit the rich data generated in ITS applications and be of great practical value especially when facing new problems with little domain knowledge.

The remainder of the paper is organized as follows. Section II briefly reviews existing works. Section III introduces the routing problems and the baseline models. Section IV presents our step-wise models in great detail. Section V provides the computational experiments and analysis. Finally, Section VI concludes the paper.

II. Related Work

Motivated by the recent advances of deep learning, researchers have been exploiting the idea of applying deep neural networks to solve challenging combinatorial optimization problems [16], using both supervised and reinforcement learning. Typical deep learning models usually have a fixed output dictionary. However, combinatorial optimization problems such as TSP and CVRP often have varying sizes, therefore seeking a deep model for solving these problems is tricky. The first deep architecture that satisfies this requirement is PtrNet [9], which resolves the issue of varying output dictionary size by introducing a pointer structure for deep learning models. It considers a combinatorial problem instance as a sequence of input nodes, and uses an LSTM Network to encode the sequence and another LSTM to decode the solution.

While PtrNet is trained by supervised learning in [9], Bello et al. [10] train PtrNet in the reinforcement learning setting using the Policy Gradient algorithm. They show that reinforcement learning is effective for training TSP solvers on larger problem sizes with better results. Nazari et al. [11] notice that the model should be invariant to the sequence of the input nodes, and replace the LSTM in the encoder of PtrNet with point-wise projection. They train this model also with Policy Gradient, and present promising results on CVRP.

Different from PtrNet and its variants in [9]–[11] that use LSTM based sequence-to-sequence models, Kool et al. [12] adopt the self-attention mechanism in the recently proposed Transformer model [14] for encoding nodes. They train this elegant model (TAM) also with reinforcement learning, using Policy Gradient with a greedy rollout baseline. On a variety of routing problems including TSP and CVRP, TAM outperforms the methods in [9]–[11] and a wide range of traditional non-learning based methods, and is the state-of-the-art deep learning model for routing problems.

With a different scheme, Dai et al. [8] use Structure2Vec [17], a graph embedding framework to embed the nodes based on the connectivity of the underlying graph. By training with deep Q-network, this model works quite well on some graph combinatorial problems. However, as empirically shown in [12], it does not perform satisfactorily for problems such as TSP and CVRP that are defined on fully connected graphs.

Except [8], all models discussed above fall into the encoder-decoder paradigm, where the encoder processes the nodes in the graph once to get the node embeddings, and the decoder selects one node at each step sequentially to get the solution. However, as we have mentioned, they do not consider the issue of removing the visited nodes that are irrelevant to the remaining sub-problems, and therefore, may result in sub-optimal policies and relatively large optimality gaps. In contrast, we explicitly eliminate the irrelevant nodes using the step-wise scheme. Even though the computational complexity raises, with elegant approximation, this scheme can yield better performance over the original models with reasonable computational overhead.

Note that it is not clear whether supervised or reinforcement learning is better for combinatorial optimization problems. On the one hand, supervised learning is more well-studied and usually converges faster. On the other hand, reinforcement learning has the advantage of not requiring true labels (i.e. optimal solutions), which could be extremely expensive to obtain for large-scale combinatorial problems. Nevertheless, as will be shown later, our step-wise scheme works well in both supervised and reinforcement learning settings.

More recently, NeuRewriter [18] uses a deep learning model to learn how to choose the region and rule for iteratively changing existing solutions. Similarly, Wu et.al [19] learn the improvement heuristic, specifically without the heavy dependence on hand-designed features for 2-opts. However, this type of model needs to iterate for a much longer time compared to construction heuristics models as ours. In another work, L2I [20] learns which different local move space to search over. At each step, it picks one type of local move and greedy searching exhaustively, which results in better solutions but a prohibitively longer time. Different from the sequential prediction task as construction or improvement heuristics, Joshi et.al [21] learn the edge prediction in a non auto-regressive manner with graph convolutional networks. However, it depends on supervision therefore can only be trained for TSP. As the optimal solutions for other routing problems such as CVRP can not be obtained in a reasonably

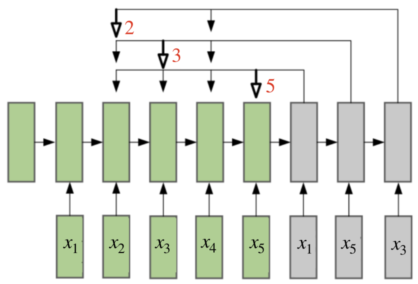


Fig. 1: PtrNet

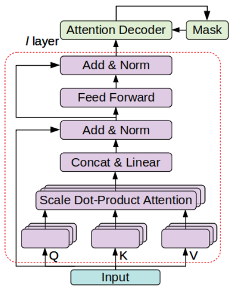


Fig. 2: TAM

long time. And it has to depend on sampling thousands of solutions to get good performance, which results in a very long solving time.

In the same time, other than routing problems, some works ([22] [23] [24]) use different kinds of graph neural networks to learn with various training schemes for combinatorial optimization problems like Satisfiability.

III. Preliminaries

A. The Routing Problems

Given a set of nodes $X = \{x_1, ..., x_n\}$ and the distances between each pair of nodes $D(x_i, x_j)$, the routing problem is defined as finding the optimal solution $Y = y_1, ..., y_m$ which is a sequence of nodes with $y_i \in X$. Usually the optimal solution is the one with the least total distance with respect to some constraints. Among all the routing problems, we consider the two most important types, i.e. TSP and CVRP.

For TSP, $n = m$ and each y_i is a distinct element in X. The goal is to minimize the tour distance, $S = \Sigma_{i=1}^{n} D(y_i, y_{i+1})$ where $y_{n+1} = y_1$, i.e. the tour should return to the starting node. CVRP is a variant of TSP with an additional node called depot x_0, and every other node has a demand $b(x_i)$ to be served by the vehicles. Multiple (M) routes could be planed, each of which (indexed by j) is a tour of a vehicle that starts from x_0, visits a subset of n_j nodes and returns to x_0. The total demands in a route j, i.e. x_0, y_1^j, ..., $y_{n_j}^j$, x_0 should not exceed the vehicle capacity c, i.e. $\Sigma_{i=1}^{n_j} b(y_i^j) \leq c$. Each $x_i \in X$ should be covered by exactly one route except x_0. The goal is to minimize the total tour distance, i.e. $S = \Sigma_{j=1}^{M} \Sigma_{i=0}^{n_j} D(y_i^j, y_{i+1}^j)$ where $y_0^j = y_{n_j+1}^j = x_0$.

B. PtrNet

Given input sequence $X = x_1, ..., x_n$, the basic sequence-to-sequence model [25] with trainable parameters θ uses two Recurrent Neural Networks (RNNs) as the encoder and decoder, respectively, to compute the conditional probability of the output sequence $Y = y_1, ..., y_m$ of length m given X based on the probability chain rule. Here we adopt the convention in [9] using the following equation to demonstrate the inference process of sequence-to-sequence models:

$$p(Y|X;\theta) = \prod_{t=1}^{m} p_\theta(y_t|y_1, ...y_{t-1}, X; \theta). \tag{1}$$

~~However, the basic sequence-to-sequence model requires a predefined output dictionary with fixed size, making it unsuitable for combinatorial optimization problems. For example, solution of a TSP instance should be a permutation of input nodes, the number of which varies in different instances. PtrNet resolves this issue by slightly modifying the basic sequence-to-sequence model with context-based attention [26]. To solve TSP, PtrNet takes the node set X as an input sequence in an arbitrary order, and employs an LSTM encoder to sequentially map the linearly projected coordinates of x_i into the hidden state of encoder LSTM h_i^e as node embeddings. Memory state of the encoder (the hidden state after encoding all nodes) is further passed into the decoder, which is another LSTM sequentially taking the last selected node y_{t-1} as input to get decoder hidden state h_t^d at step t. The conditional distribution over the next node at step t is modeled as the pointer attention:~~ To solve TSP, PtrNet employs an LSTM encoder to sequentially map the linearly projected coordinates of x_i into the hidden state of encoder LSTM h_i^e as node embeddings. The decoder of PtrNet is another LSTM with the memory state passed from the encoder (the hidden state after encoding all nodes). It sequentially takes the last selected node y_{t-1} as input to get decoder hidden state h_t^d at step t and outputs the conditional distribution over the next node at step t as the pointer attention:

$$u_i^t = v^T \tanh(W_1 h_i^e + W_2 h_t^d), \quad i \in U_t, \tag{2}$$

$$p_\theta(y_t|y_1, ...y_{t-1}, X; \theta) = \text{softmax}(u^t), \tag{3}$$

where v, W_1 and W_2 are learnable parameters, and U_t is the set of unvisited nodes, i.e., the feasible selections at step t. Here, u_i^t is the ith element of vector u_t with dimension $|U_t|$, which is further processed by a softmax function in Eq. (3). ~~The starting node y_1 is always the first node x_1 in the sequence, and the current (last selected) node y_{t-1} is the input to the decoder at step t.~~ The process of PtrNet solving a TSP instance with 5 nodes is depicted in Fig. 1. The encoder LSTM (green boxes) encodes five nodes sequentially. And with the starting node y_1 always being the first node x_1 in the sequence, the decoder LSTM (grey boxes) decodes three steps to select x_5, x_3 and x_2. Since the only unvisited node is x_4, a solution is found, i.e. the route $x_1 \to x_5 \to x_3 \to x_2 \to x_4 \to x_1$.

C. TAM

Different from the sequence-to-sequence models, the Transformer Attention Model (TAM) [12] ignores the recurrence structure and is designed solely based on the dot-product attention. For TSP, the input to TAM is the two-dimensional coordinates of nodes, while an additional dimension *demand* is included for CVRP. The encoder follows the architecture of Transformer [14], which is a linear projection $X_i^p = W_p x_i$ followed by multiple (L) layers of self-attention blocks. ~~These building blocks keep the output features in the same dimension as the input, thus can be stacked together.~~ The core of self-attention block is a multi-head attention layer:

$$Q_i^h = W_Q^h X_i^p, \quad K_i^h = W_K^h X_i^p, \quad V_i^h = W_V^h X_i^p \tag{4}$$

$$A^h = \text{softmax}(Q^h {K^h}^T / \sqrt{d_k}) V^h, \text{for } h = 1, 2, ..., H, \tag{5}$$

$$\text{Multihead}(Q, K, V) = \text{Concat}(A^1, A^2, ..., A^H) W_O, \tag{6}$$

where X^p is the input to this layer, i is node index, h is head index and H is the number of heads. $W_Q^h, W_K^h, W_V^h \in \mathbb{R}^{d \times d_k}$, $W_O \in \mathbb{R}^{d \times d}$ are trainable parameters, d is the feature dimension, $d_k = d/H$. Here, Q, K, V are linear projections of node embeddings and represent the Query, Key, Value vector according to the rationale of self-attention [14]. The self-attention block works as follows:

$$\hat{f}_i = BN(X_i + \text{Multihead}_i(Q, K, V)), \tag{7}$$

$$f_i = \text{SelfAttention}_i(X^p) = BN(\hat{f}_i + \text{FF}(\hat{f}_i)), \tag{8}$$

where f is output of the self-attention block, FF is two feed-forward layers with ReLu activation. Both (7) and (8) has a skip-connection [27] and a Batch Normalization layer [28].

The decoder models $P(y_t|X, y_1, ..., y_{t-1})$ using an attention mechanism as follows: ~~Query of the dot-product attention comes from a context embedding $f_c = \text{Concat}(\bar{f}, f_{y_0}, f_{y_{t-1}})$, i.e. the concatenation of the mean of node embeddings $\bar{f}$, the embedding of the starting nodes f_{y_0} and current node $f_{y_{t-1}}$. For the first step, embeddings of the starting and current nodes are replaced by trainable parameters. This process is described as follows:~~

$$f_c = \text{Concat}(Mean(f), f_{y_0}, f_{y_{t-1}}) \tag{9}$$

$$f_c^g = \text{Multihead}(W_Q^g f_c, W_K^g f^t, W_V^g f^t), \tag{10}$$

$$q = W_Q f_c^g, k_i = W_K f_i, \tag{11}$$

$$o_i = C \tanh(q^T k_i / \sqrt{d}), i \in U_t, \tag{12}$$

$$P(y_t|X, y_1, ..., y_{t-1}) = \text{softmax}(o), \tag{13}$$

where f^t is node embeddings, d is feature dimension. Here, o is similar to u^t in (3). In (9), for the first step, f_{y_0} and $f_{y_{t-1}}$ are replaced by trainable parameters. And (10) is similar to the glimpse in [29] with a mulit-head attention to get a new context. $C = 10$ is used to clip o_i to [-10,10] for better exploration [10]. The structure of TAM is shown in Fig.2, where the purple and green boxes are the operations running only once and at every decoding step, respectively.

IV. Models

In this section, we describe our approach in detail. We first show how to apply the step-wise scheme to PtrNet and TAM. Then, we present the approximate SW-TAM which can reduce the computational overhead of the step-wise TAM.

A. Step-Wise PtrNet

Clearly in PtrNet, $p_\theta(y_t|y_1, ...y_{t-1}, X; \theta)$ is a function of all nodes in the input X. In other words, while selecting one node to visit, the node embeddings depend on the whole graph including the already visited nodes. However, the previous decisions in the solution construction process cannot be altered and should not affect the future decisions. Therefore with the irrelevant information of the visited nodes for the embeddings, the model may not produce an optimal policy for the sub-problems at each step. Inspired by the Bellman's Principle of Optimality, we correct this by explicitly eliminating the visited and irrelevant nodes in the graph at each step. More specifically, after one node is selected and visited at step t, the encoder LSTM re-embeds the remaining nodes with this visited node eliminated from the sequence to get new embeddings e_i^t, instead of simply using the original embeddings e_i. Note that we do not remove the default starting node x_1, since the salesman needs to return to x_1 eventually. We call this model step-wise PtrNet (SW-PtrNet).

Here we demonstrate the inference process of SW-PtrNet, i.e. how to re-embed the node information and select the next node. In each decoding step t, the encoder LSTM first embeds each unvisited node x_i sequentially. The resulting embedding for x_i is the ith hidden state in the encoder $e_i^t = h_i^e = \text{LSTM}_e(x_i, h_{i-1}^e)$. After that, the decoder computes its hidden state as $h_t^d = \text{LSTM}_d(y_{t-1}, h_t^m)$, where y_{t-1} is the node selected in step $t-1$ and h_t^m is the memory state from the encoder in step t (note that the encoder and decoder do not share parameters). Then, the conditional distribution over the next node y_t can be obtained as follows:

$$u_i^t = v^T \tanh(W_1 e_i^t + W_2 h_t^d), \quad i \in U_t, \tag{14}$$

$$p_\theta(y_t|y_{t-1}, y_1, V_t; \theta) = \text{softmax}(u^t). \tag{15}$$

While solving TSP, the model keeps track of the visited nodes and masks them out for the probability of selecting y_t, i.e., only the nodes $i \in U_t$ (the set of unvisited nodes) can be selected at step t. Therefore, the constraint for not visiting the same nodes twice is satisfied. After selecting y_t based on the above distribution, SW-PtrNet moves to the next step. The visited node y_t is removed from the input sequence and the encoder LSTM is used to re-embed the sequence to get e_i^{t+1} for the decoder at step $t+1$. Therefore, different from h_i^e in (2), the node embeddings e_i^t in (14) is re-embedded at each step. As a result, the conditional probability in (15) no longer conditions on the previous visited nodes, unlike (3). Since the decoder only decodes one node during each step t, there is no need to maintain its memory state. The process is repeated until only one node is unvisited. Then a tour is obtained by appending the unvisited node and the starting node to the sequence, such that the constraints of TSP are satisfied. The process of SW-PtrNet solving a TSP instance with 5 nodes is shown in Fig. 3.

The algorithm for solving one TSP instance using SW-PtrNet is shown in Algorithm 1. With LSTM re-embedding the nodes at each step, the computational complexity of the encoder at inference time raises one order up from $O(nd^2)$ to $O(n^2d^2)$, where d is the feature dimension and n is the

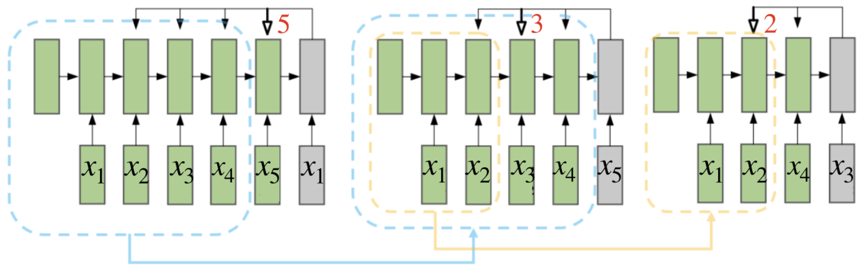


Fig. 3: SW-PtrNet

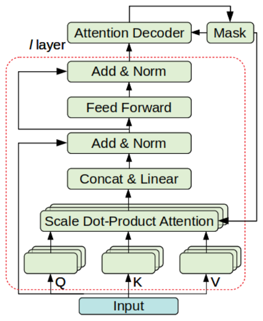


Fig. 4: SW-TAM

```
Data: Set of nodes $X$
Result: The solution sequence $Y$
1 Embedding sequence with all the nodes using the
    encoder Using the encoder LSTM;
2 while not all nodes are visited do
3     Select the node to visit using the decoder LSTM
        with (14) and (15);
4     Eliminate the selected node and re-embed the
        unvisited nodes using the encoder LSTM;
5 end
```

Algorithm 1: Inferring a Solution Using SW-PtrNet

number of nodes. This is due to Line 4 in Algorithm 1 which has worst case computation complexity of $O(nd^2)$ for each of the n runs. Intuitively, for each step t the encoder needs to eliminate y_{t-1} and re-embed all the remaining unvisited nodes. However, we can save some computation by reusing some of the previous embeddings. In fact, the encoder LSTM just needs to truncate right before $x_s = y_{t-1}$ in the input sequence of step $t-1$, and embed the nodes after x_s. To achieve this, we keep the first $s-1$ embedding steps in the encoder and skip the selected node x_s. After that, we embed node x_{s+1} as $e^t_{s+1} = h^e_{s+1} = \text{LSTM}_e(x_{s+1}, h^e_{s-1})$, and other unvisited ones similarly. We illustrate this process in Fig. 3, where the dashed boxes are the embeddings that can be reused. Since a significant amount of computation could be saved, we do not seek an approximation for SW-PtrNet.

B. Step-Wise TAM

The original TAM model also suffers from the same issue as PtrNet, i.e. the encoder embeds all nodes only once, and the same embeddings are used throughout decoding, without removing previously visited nodes that are irrelevant to future decisions. Therefore, the embeddings calculated at the beginning are no longer accurate after a couple of steps, resulting in sub-optimal policies. Although the glimpse mechanism in (10) tries to correct the context for the query vector q, the node embeddings to form this query vector and the key vectors k_i still correlate with the irrelevant information from the visited nodes. To fix this, we apply the step-wise scheme to TAM, and explicitly make the embeddings independent of the irrelevant nodes. To this end, at each step, the Transformer encoder re-embeds the whole graph with the irrelevant nodes eliminated. More specifically, we project the nodes linearly with $X^p_i = W_p x_i$ and run the whole L layers of self-attention block as (8) in the encoder at each step t with the irrelevant nodes masked:

$$f^t_i = \text{SelfAttention}^t_i(X^p). \tag{16}$$

In the multi-head attention of (16), to mask the irrelevant nodes, we compute the attention compatibility A^h using the following equation instead of (5):

$$w^h_{ij}(Q^h_i, K^h_j) = \begin{cases} Q^h_i {K^h_j}^T/\sqrt{d_k}, & j \in U_t \\ -\infty, & \text{otherwise}, \end{cases} \tag{17}$$

$$A^h = \text{softmax}(w^h)V^h. \tag{18}$$

Here, w^h_{ij} is the jth element of the ith vector of the matrix w^h with dimension $n \times n$ and the softmax in (18) is along the second dimension. As shown in Algorithm 2, after selecting one node to visit at step t (line 3), the encoder re-embeds the new set of feasible nodes $i \in U_t$ with (16)-(18), and the decoder takes the new embeddings for step $t+1$ (line 4). Note that for TSP, the set of feasible nodes U_t consists of the unvisited nodes; while for CVRP, U_t consists of unvisited nodes which also satisfy the capacity constraints. The model keeps track of the history of selected nodes to get the set of feasible nodes U_t for each step t. And the decoder decodes the probability of selecting next nodes based on the new node embeddings where the probability is masked out for the infeasible nodes $i \notin U_t$. Therefore after all nodes are visited, a feasible solution that satisfies all the constraints for the routing problems (TSP or CVRP) can be obtained.

```
1 Embedding the set of nodes using the Transformer
    Encoder with (4) - (8) ;
2 while not all nodes are visited do
3     Select the node to visit using the Attention
        Decoder with (9) - (13);
4     Eliminate the infeasible nodes and re-embed the
        nodes with (16) - (18);
5 end
```

Algorithm 2: Inferring a Solution Using SW-TAM

We call the above model step-wise TAM (SW-TAM), which is no longer a model for the whole problem but for each sub-problem of selecting one node to visit. Its structure is illustrated in Fig. 4 following the same style of Fig. 2. Both TAM

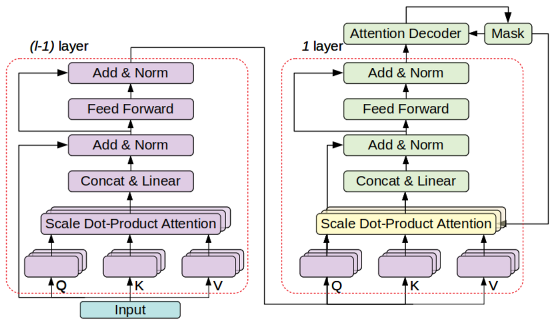


Fig. 5: ASW-TAM

and SW-TAM are trained in the reinforcement learning setting, where the model parameters are optimized to maximize the cumulative reward that corresponds to minimizing the total tour length. Following [12], we use standard REINFORCE algorithms with a greedy roll-out baseline. Specifically, the baseline policy is the periodically saved best performing model parameters. After evaluating a held-out dataset at the end of each epoch, the best set of parameters is kept to greedily infer the result as the baseline.

In Algorithm 2, similar to SW-PtrNet, the re-embedding operation (Line 4) also introduces additional overhead to SW-TAM. This is due to the fact that the problem with n nodes breaks into sub-problems in each step, where the encoder needs to encode the nodes n times. The computation complexity of the multi-head self-attention block ((16)) is $O(nd^2+dn^2)$ if we consider the number of heads as constant [14]. Due to (16) running for n times, the computational complexity of the encoder in SW-TAM is $O(n^2d^2+dn^3)$, which is cubic of the number of nodes and hard to scale up to larger graphs. In the next subsection, we design an approximation of SW-TAM that is more computationally efficient.

C. Approximate SW-TAM

It is well-known that in Convolutional Neural Network for face recognition, the lower convolutional layer extracts more general features such as lines in different degrees. On the contrary, the upper layer extracts specific features that are more direct for performing the task such as detecting different types of eyes or noses [30]. Based on this intuition, we hypothesize a similar feature extraction paradigm for the attention model, which we will validate in the experiments. Below we employ it to achieve better computational efficiency.

More specifically, we propose a new model which approximately performs the re-embedding process in SW-TAM by fixing the lower attention layer at each step, and change only the top layer by masking out the attention weights of irrelevant nodes in the multi-head attention mechanism. In this new model with L multi-head attention blocks, the first $L-1$ is the same as standard building blocks, which can be viewed as the encoder. On the other hand, the top attention layer is part of the decoder where at each step the attention mechanism in this layer masks out the irrelevant nodes and gets new embeddings. This design is illustrated in Fig. 5, where the purple boxes are the encoder running only once for embeddings, and the green boxes are the decoder running at each step. The yellow box (multi-head attention layer in the last attention block) is also part of the decoder with the attention weights pre-computed for once and masked at every step. The first part (pre-computation) of the yellow box is defined mathematically as (4) and the following equation:

$$~~Q_i^h = W_Q^h X_i^p, K_i^h = W_K^h X_i^p, V_i^h = W_V^h X_i^p,~~ \quad (19)$$

$$w^h(Q^h, K^h) = Q^h K^{h^T} / \sqrt{d_k}, \quad (20)$$

The second part of the yellow box runs at each step t, and is to set $w^h_{\cdot, C_{t-1}} = -\infty$ and perform the following computation and (6):

$$A^h = \text{softmax}(w^h)V^h, \quad (21)$$

$$~~\text{Multihead}(Q, K, V) = \text{Concat}(A^1, A^2, ..., A^H)W_O.~~ \quad (22)$$

Then the node embeddings are computed using (7) and (8). Based on the new embeddings, the rest part of the decoder selects a new node and gets a new mask. The model will then move to step $t+1$ and runs the second part in the yellow box with the new mask. We call this model Approximate SW-TAM (ASW-TAM). The inferring process for ASW-TAM is shown as in Algorithm 3. The constraints can be satisfied using the same procedure as that for SW-TAM.

```
1 Embedding the set of nodes using the Transformer
    Encoder with (4) - (8) ;
2 while not all nodes are visited do
3     Select the node to visit using the Attention
        Decoder with (9) - (13);
4     Eliminate the infeasible nodes and re-embed the
        nodes approximately with (21) and (6);
5 end
```
Algorithm 3: Inferring a Solution Using ASW-TAM

In this new proposed layer in ASW-TAM, (4) and (20) only need to run once and have the computational complexity of $O(nd^2)$ and $O(dn^2)$ respectively. (4), (20) and the first $L-1$ attention layers run in Line 1 in Algorithm 3. (6), (7) and (8) need to run every decoding step as Line 4 in Algorithm 3 resulting in $O(n^2d^2)$ computational complexity in total. Note that straightforward implementation of (21), which will be performed n times, results in the complexity of $O(n^3d)$. However, with elegant engineering, we managed to achieve $O(n^2d)$ for calculation of (21) as shown in the following.

We omit the head index h for simplicity, and (21) can be explicitly rewritten into element-wise form as follows,

$$\hat{w}_{i,j} = \exp(w_{i,j}), \text{ for all } i \text{ and } j, \quad (23)$$

$$A_{i,j} = \frac{\sum_k \hat{w}_{i,k} V_{k,j}}{\sum_k \hat{w}_{i,k}}, \text{ for all } i \text{ and } j. \quad (24)$$

And we define

$$sw_i(t=1) = \sum_k \hat{w}_{i,k}, \text{ for all } i, \quad (25)$$

$$swV_{i,j}(t=1) = \sum_k \hat{w}_{i,k} V_{k,j}, \text{ for all } i \text{ and } j. \quad (26)$$

(23), (25), and (26) are pre-computed once and have the computation complexity of $O(n^2 d)$. At step $t = 1$, none of the nodes is masked and each element of A is calculated as follows,

$$A_{i,j} = \frac{swV_{i,j}(1)}{sw_i(1)}, \text{ for all } i \text{ and } j. \quad (27)$$

Then at step $t-1$ after decoding, the node with index l is visited and should be removed. Accordingly, each element of A at step t is calculated as follows,

$$sw_i(t) = sw_i(t-1) - \hat{w}_{i,l}, \text{ for all } i, \quad (28)$$

$$swV_{i,j}(t) = swV_{i,j}(t-1) - \hat{w}_{i,l} V_{l,j}, \text{ for all } i \text{ and } j, \quad (29)$$

$$A_{i,j} = \frac{swV_{i,j}(t)}{sw_i(t)}, \text{ for all } i \text{ and } j. \quad (30)$$

For each element, the computational complexity with respect to each of (27), (28), (29) and (30) is $O(1)$ and the total number of elements is $O(nd)$. Therefore, the computational complexity for each step is $O(nd)$, and for all the steps is $O(n^2 d)$. Consequently, the total computational complexity of the top attention layer is considered as $O(n^2 d^2)$. Note that the equations are written in the element-wise form only for illustration purpose.

Compared to TAM, ASW-TAM raises the computation complexity from $O(nd^2 + dn^2)$ to $O(n^2 d^2)$. However, it is still quadratic in the number of nodes. Even d could be large as 128, we only need to do this in the top layer, which turns out to be acceptable as demonstrated in the experiments.

V. Experiments

In this section, we conduct experiments to verify the effectiveness of our step-wise deep learning models on two types of routing problems, i.e. TSP and CVRP.

Following [11], [12], here the locations of TSP and CVRP instances are generated independently from a two-dimensional uniform distribution ranging from 0 to 1 for both dimensions. We generate instances with 20, 50, 100 nodes for TSP and CVRP. For convenience, we refer them as TSP20, CVRP20, etc. For CVRP, the demands of each node are sampled uniformly from discrete numbers $\{1...9\}$ while the vehicle capacities are fixed as 30, 40, 50 for CVRP20, CVRP50, CVRP100, respectively. For testing, we generate 10,000 samples from the same distribution and keep it fixed for all models. For training and evaluation, we use the highly optimized solver Concorde [31] to get the optimal solutions for TSP. For CVRP, since it is much harder to be solved optimally, we use the state-of-the-art solver LKH3 [32] to get the benchmark solutions.

A. SW-PtrNet

In this subsection, we compare SW-PtrNet with the original PtrNet. For a fair comparison, we adopt a similar training scheme and problem settings as in [9], i.e. using supervised training on TSP problems. We use the TSP20 instances for training and testing, as it is the largest training size in [9]. Moreover, with larger problem sizes, the availability of the optimal solutions as training samples is at great expense due to the exponential computational complexity of the exact algorithms. Therefore, we generate 1,600,000 TSP20 instances, and solve them using Concorde to get the optimal solutions for training. On the other hand, the exact algorithm for CVRP is far less efficient than that for TSP, and it is prohibitively time-consuming to generate sufficient optimal solutions as training samples. Therefore, we do not consider CVRP for PtrNet or SW-PtrNet in this subsection.

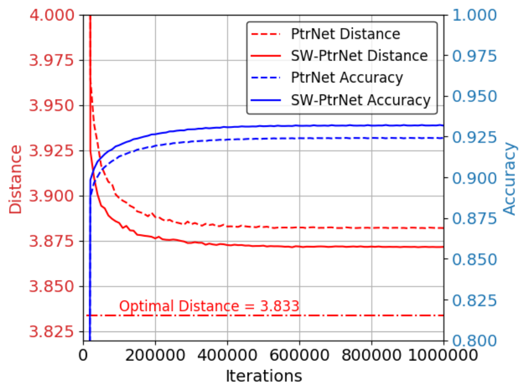


Fig. 6: PtrNet vs SW-PtrNet during training

TABLE I: PtrNet vs SW-PtrNet

Models	Distance	Opt. Gap	Accuracy	Total Time
PtrNet	3.882	1.28%	92.39%	**3s**
SW-PtrNet	**3.872**	**1.00%**	**93.17%**	14s

The models are trained with the Adam Optimizer [33] where the learning rate is initially 0.001 and decayed by 0.96 every 5,000 steps. We train both models for 1,000,000 iterations with batch size of 128 and clip the gradient by 2.0 of L_2 norm. The coordinates of nodes are projected into a 256 dimension space before being fed into the decoder. The hidden dimension for both the encoder and decoder is 256. The parameters are initialized with uniform distributions in the range of [-0.08, 0.08]. For TSP, since the route is a loop, the selection of the first node does not matter and is assumed to be known. Thus for each sampled instance, only 18 selections need to be predicted. During the training for both models, we mask out the output for infeasible selections as shown in (14).

Here we use four measures for evaluation, including the average solution distance, optimality gap, and total inference time on the 10,000 testing instances, as well as training accuracy. We use accuracy here because both models are trained by supervised learning to imitate the construction of optimal (training) solutions. In such setting, accuracy can directly reflect the ability of a model in terms of imitation. We only report training accuracy because it is more informative. Since there exist at least two optimal routes for each TSP instance, testing accuracy could be very low for one of the optimal routes as it can easily be different from the true label. This mismatch is much less severe for training accuracy due to the fixed inputs to the decoder.

In Fig. 6, we plot the performance of PtrNet and SW-PtrNet during training, measured by the testing distance and training

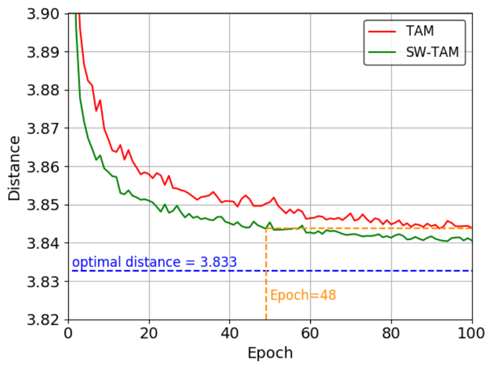


Fig. 7: TAM vs SW-TAM for TSP 20

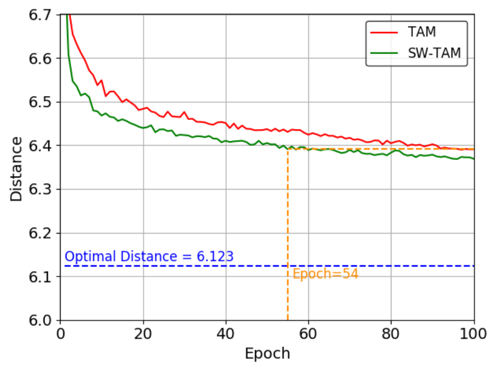


Fig. 8: TAM vs SW-TAM for CVRP 20

accuracy after every 1,000 training steps. We can observe from Fig. 6 that SW-PtrNet almost consistently outperforms PtrNet during training in terms of both measures. Training accuracy indeed indicates good performance here, since SW-PtrNet which is more accurate consistently produces lower solution distance. We further list the convergence performance in Table I, along with the average optimality gap and the total inference time for the 10,000 testing instances. SW-PtrNet clearly outperforms PtrNet in terms of solution quality. Regarding the optimal gap, the relative improvement for SW-PtrNet over PtrNet is 21.88%. The inference time of SW-PtrNet is longer than PtrNet, which is not surprising since it has higher computational complexity as we have analyzed in Section IV. Nevertheless, both models are very fast since they solve 10,000 instances within 15s.

To summarize, our step-wise scheme can effectively improve the performance of the original PtrNet in solving TSP.

B. SW-TAM

In this subsection, we show the effectiveness of SW-TAM by comparing with the original TAM. As mentioned, SW-TAM is computationally expensive, therefore we use small routing problems here, i.e. TSP20 and CVRP20. For a fair comparison, we use the same setting and hyper-parameters as those in the original TAM, and train it using REINFORCE with the greedy rollout baseline for 100 epochs [12]. Each epoch has 1,280,000 instances generated on the fly.

TABLE II: ASW-TAM Results vs TAM Results (values in the parentheses are the total inference time for solving the 10,000 testing instances)

Problem	Opt. Obj.	TAM Obj.	TAM Gap	ASW-TAM Obj.	ASW-TAM Gap	Impr. on Gap
TSP20	3.83	3.84 (0s)	0.29%	**3.84** (1s)	**0.23%**	20.69%
TSP50	5.69	5.79 (2s)	1.79%	**5.76** (7s)	**1.16%**	35.19%
TSP100	7.76	8.10 (6s)	4.35%	**8.01** (29s)	**3.20%**	26.39%
CVRP20	6.12	6.40 (1s)	4.46%	**6.39** (2s)	**4.29%**	3.66%
CVRP50	10.38	10.98 (3s)	5.82%	**10.90** (9s)	**4.98%**	14.40%
CVRP100	15.64	16.75 (8s)	7.07%	**16.42** (39s)	**4.99%**	29.44%

We plot the training curves of greedy decoding TAM and SW-TAM on TSP20 and CVRP20 in Figs. 7 and 8, respectively. As can be observed, SW-TAM is significantly better than the original TAM on both problems. In terms of optimality gap for TSP and CVRP, SW-TAM reduces those of original TAM from 0.29% to 0.19% and 4.46% to 4.15%, respectively. Moreover, it takes only 48 and 54 epochs for SW-TAM to reach the results of TAM after 100 epochs.

However, the computational complexity of re-embedding after each step in SW-TAM is one order higher than that of the original model. Regarding TSP and CVRP, it takes 52 and 66 minutes to train an epoch for SW-TAM, while TAM only needs 6 and 9 minutes, respectively. Hence it is hard to scale up to large problems with 50 or 100 nodes due to the memory and time constraints. This motivates us to design the approximate SW-TAM model, which will be evaluated in the next subsection.

C. Approximate SW-TAM

In this subsection, we compare the performance of ASW-TAM with the original TAM on TSP and CVRP instances with 20, 50, and 100 nodes. The training settings are the same as those in the previous subsections.

The results are summarized in Table II. Note that similar to Figs. 7 and 8, these results are based on greedily decoding the model, which is sufficient to prove the effectiveness of our proposed step-wise scheme. However, the solution quality of our model can also be further improved if additional techniques such as sampling or beam search are applied. As can be observed, ASW-TAM consistently outperforms the original TAM model in terms of distance and optimality gap on all instances. The relative improvement is significant for most of these problems. Moreover, the improvement on the optimality gap tends to be larger on harder instances with more nodes. Though the inference time of ASW-TAM is longer than TAM due to the computational overhead of re-embedding the nodes approximately, it is still very fast and acceptable as it takes less than one minute to solve 10,000 instances with 100 nodes. Compared to traditional non-learning solvers based on complicated search algorithms, the inference time is still very short and even negligible. For example, LKH [34] takes 13 hours to solve 10,000 instances of CVRP100 with 32 parallel threads as shown in [12].

In terms of training efficiency, on TSP20, it takes only 12 minutes for ASW-TAM to train an epoch, which is

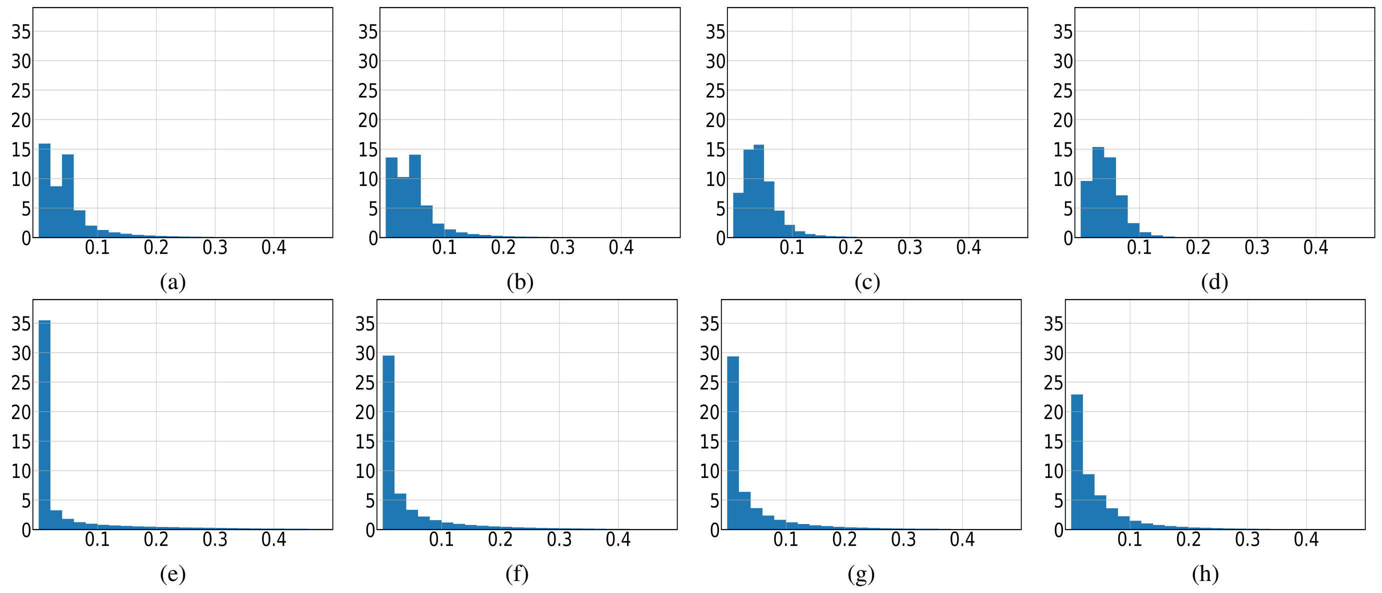


Fig. 9: Histograms of Attention Weights in TAM and ASW-TAM for TSP20 and CVRP20, the coordinates of x-axis and y-axis for each bin are the values of attention weights and the percentage frequencies divided by the bin width, respectively. (a) TSP, TAM, bottom. (b) TSP, ASW-TAM, bottom. (c) CVRP, TAM, bottom. (d) CVRP, ASW-TAM, bottom. (e) TSP, TAM, top. (f) TSP, ASW-TAM, top. (g) CVRP, TAM, top. (h) CVRP, ASW-TAM, top

significantly less than the 52 minutes of SW-TAM, and is acceptable compared to the 6 minutes for the original TAM. This drastic improvement enables us to train ASW-TAM on larger instances with 50 and 100 nodes. For TSP20, the optimality gap of ASW-TAM is 0.23%, which is a bit larger than that of SW-TAM (0.19%). This shows that ASW-TAM trades off performance for better computational efficiency, which validates the effectiveness of our design.

To better understand the rationale of how ASW-TAM works, we plot the weights of the bottom and top attention layers of both TAM and ASW-TAM in solving TSP20 and CVRP20 using histograms in Fig. 9, respectively. We can observe that the pattern of attention weights distribution is clearly different across different layers. More specifically, weight distribution in the bottom layer (layer 1) is flatter in the sense that the attention weights are mostly around 0.05, which is the average weight for 20 nodes, and almost all of the attention weights are within 0.3. In contrast, the distribution of the weights in the top layer (layer 3) is more uneven where 30 percent of the weights are less than 0.02 and the attentions with larger weights are much more than the corresponding proportion of the bottom layer. This observation validates our intuition in Section IV-C. In particular, the lower attention layer extracts more fundamental features of the input instance. On the other hand, the upper layer extracts features that are more directly helpful for solving the problem based on the uneven weight distribution, and is much more aggressive in the sense that it will pay more attention to the informative nodes and barely any attention to irrelevant nodes. Therefore, we keep the embeddings from lower layers unchanged in ASW-TAM, and only perform re-embedding in the top layer at each decoding step to obtain features that are more related to solving the problems by masking out the visited nodes.

Following [8], we further compare the performance of our model ASW-TAM and the original TAM on 38 instances

TABLE III: ASW-TAM Results vs TAM Results on TSPLIB (number in the instance name is its size)

Instance	Opt.	TAM		ASW-TAM		Impr. on
name	Obj.	Obj.	Gap	Obj.	Gap	Gap
berlin52	7542	8898	17.98%	**8229**	**9.11%**	49.31%
st70	675	700	3.69%	**694**	**2.80%**	24.17%
kroA100	21282	26035	22.33%	**24424**	**14.76%**	33.89%
rd100	7910	8490	7.33%	**8090**	**2.27%**	69.04%
pr107	44303	51735	16.77%	**45625**	**2.98%**	82.21%
kroA150	26524	30356	14.45%	**28964**	**9.20%**	36.32%
kroA200	29368	35623	21.30%	**34194**	**16.43%**	22.85%
tsp225	3916	4738	21.00%	**4709**	**20.25%**	3.57%
pr264	49135	61060	24.27%	**54736**	**11.40%**	53.03%
pr299	48191	65320	35.54%	**63902**	**32.60%**	8.28%

from a widely used benchmark TSPLIB [35], ranging from 51 to 318 cities. We directly perform inference on these instances using the models trained for TSP100. To get a better generalization performance on these instances with different sizes, the statistics for the Batch Normalization layer in both models come from instances with the same number of nodes as the one being inferred, which can be computed in advance and introduces no additional computational overhead. Results show that ASW-TAM outperforms TAM with an optimal gap of 12.20% against 14.75%, and finds better solutions for over 76% (29 out of 38) instances. Table III shows results on 10 selected instances due to limited space. We can observe that ASW-TAM performs well across various problem sizes. The total inference time of the 38 instances for TAM and SW-TAM are 0.56s and 0.89s respectively.

TAM, SW-TAM and ASW-TAM are all based on the architecture of Transformer. However, it is known that training Transformer model could be expensive and may take a significantly long time, even with supervised learning [14]. This has more impacts on the reinforcement learning setting, since the training might converge slowly. Consequently, the

100 training epochs we adopted from [12] previously to train these three models may not yield the best performance. Hence, we train these models for a longer time and notice that they can converge to better results. However, ASW-TAM still shows consistently better performance than the original TAM. Specially, we demonstrate this using TSP20 and TSP50. In TSP20, after 600 training epochs, TAM converges to an average distance of 3.838 with optimality gap of 0.14%, and ASW-TAM converges to 3.837 with optimality gap of 0.10%. For TSP50, TAM takes 1,000 epochs to converge to an average distance of 5.739 with optimality gap of 0.82%, while ASW-TAM converges after 800 epochs to an average distance of 5.722 with optimality gap of 0.52%. The relative improvements on the optimality gap are 27% and 37% respectively, which are roughly the same as the result for 100 epochs in Table II. All our experiment codes will be made available soon.

VI. Conclusion and Future Work

In this paper, we propose a step-wise scheme for the deep models that learn construction heuristics to solve routing problems. In this scheme, the nodes that have been visited in the construction process are explicitly eliminated, such that the following sub-problems are correctly represented. We then apply this scheme to PtrNet and TAM, two representative deep models, and obtain significant improvements in terms of optimality. To ameliorate the expensive computation caused by the re-embedding operations in SW-TAM, we further propose the Approximate SW-TAM by modifying the last attention layer, which significantly reduces the complexity and retains good empirical performance. For future work, we would like to apply this step-wise scheme to other models and on other problems. Also, we hope that this idea would inspire further researches on designing network architectures considering the recursive nature of sequential decision making problems.

Acknowledgment

The research was supported by both the ST Engineering-NTU Corporate Lab and the Singtel-NTU Cognitive & Artificial Intelligence Joint Lab through the NRF corporate lab@university scheme. Wen Song was partially supported by the Young Scholar Future Plan of Shandong University (Grant No. 62420089964188). Zhiguang Cao was partially supported by the National Natural Science Foundation of China (61803104) and Singapore National Research Foundation (NRF-RSS2016004).

References

[1] K.-H. N. Bui and J. J. Jung, "Aco-based dynamic decision making for connected vehicles in iot system," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 10, pp. 5648–5655, 2019.

[2] Y. Feng, B. Hu, H. Hao, Y. Gao, Z. Li, and J. Tan, "Design of distributed cyber–physical systems for connected and automated vehicles with implementing methodologies," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 9, pp. 4200–4211, 2018.

[3] M. Veres and M. Moussa, "Deep learning for intelligent transportation systems: A survey of emerging trends," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–17, 2019.

[4] D. Li, L. Deng, Z. Cai, B. Franks, and X. Yao, "Intelligent transportation system in macao based on deep self-coding learning," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3253–3260, 2018.

[5] X. Wang, T.-M. Choi, H. Liu, and X. Yue, "Novel ant colony optimization methods for simplifying solution construction in vehicle routing problems," *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 11, pp. 3132–3141, 2016.

[6] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *The traveling salesman problem: a computational study*. Princeton university press, 2006.

[7] M. Fischetti, P. Toth, and D. Vigo, "A branch-and-bound algorithm for the capacitated vehicle routing problem on directed graphs," *Operations Research*, vol. 42, no. 5, pp. 846–859, 1994.

[8] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 6348–6358.

[9] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 2692–2700.

[10] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," in *Proceedings of International Conference on Learning Representations (ICLR).*, 2016.

[11] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác, "Reinforcement learning for solving the vehicle routing problem," in *Advances in Neural Information Processing Systems*, 2018, pp. 9839–9849.

[12] W. Kool, H. van Hoof, and M. Welling, "Attention, learn to solve routing problems!" in *Proceedings of International Conference on Learning Representations (ICLR).*, 2018.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[15] R. E. Bellman, "Dynamic programming treatment of the traveling salesman problem," 1961.

[16] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *arXiv preprint arXiv:1811.06128*, 2018.

[17] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *International conference on machine learning*, 2016, pp. 2702–2711.

[18] X. Chen and Y. Tian, "Learning to perform local rewriting for combinatorial optimization," in *Advances in Neural Information Processing Systems*, 2019, pp. 6281–6292.

[19] Y. Wu, W. Song, Z. Cao, J. Zhang, and A. Lim, "Learning improvement heuristics for solving routing problems," 12 2019.

[20] S. Y. Hao Lu, Xingwen Zhang, "A learning-based iterative method for solving vehicle routing problems," in *Proceedings of International Conference on Learning Representations (ICLR).*, 2020.

[21] 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.

[22] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a sat solver from single-bit supervision," *arXiv preprint arXiv:1802.03685*, 2018.

[23] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," in *Advances in Neural Information Processing Systems*, 2018, pp. 539–548.

[24] P.-W. Wang, P. L. Donti, B. Wilder, and Z. Kolter, "Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver," *arXiv preprint arXiv:1905.12149*, 2019.

[25] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[26] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proceedings of International Conference on Learning Representations (ICLR).*, 2015.

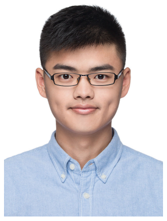
[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

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

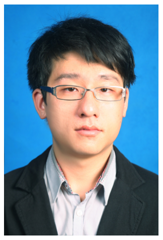
[29] O. Vinyals, S. Bengio, and M. Kudlur, "Order matters: Sequence to sequence for sets," in *Proceedings of International Conference on Learning Representations (ICLR).*, 2015.

[30] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[31] D. Applegate, R. Bixby, V. Chvatal, and W. Cook, "Concorde tsp solver," *URL http://www.math.uwaterloo.ca/tsp/concorde*, 2006.
[32] K. Helsgaun, "An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems," *Roskilde: Roskilde University*, 2017.
[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of International Conference on Learning Representations (ICLR).*, 2014.
[34] K. Helsgaun, "An effective implementation of the lin–kernighan traveling salesman heuristic," *European Journal of Operational Research*, vol. 126, no. 1, pp. 106–130, 2000.
[35] G. Reinelt, "Tsplib—a traveling salesman problem library," *ORSA journal on computing*, vol. 3, no. 4, pp. 376–384, 1991.



Liang Xin received the bachelor's degree from Tongji University and the master's degree from Carnegie Mellon University, where he is currently pursuing the phd's degree with the School of Computer Science and Engineering, Nanyang Technological University, Singapore (NTU). His research interests include deep learning for combinatorial optimization problems.



Wen Song received the B.S. degree in automation and the M.S. degree in control science and engineering from Shandong University, China, in 2011 and 2014, respectively, and the Ph.D. degree in computer science from the Nanyang Technological University, Singapore, in 2018. He was a Research Fellow in the Singtel Cognitive and Artificial Intelligence Lab for Enterprises (SCALE@NTU). He is currently an Associate Professor with the Institute of Marine Science and Technology, Shandong University, China.

His current research interests include artificial intelligence, machine learning, planning and scheduling, multi-agent systems, and operations research.

Zhiguang Cao received the Ph.D degree from Interdisciplinary Graduate School, Nanyang Technological University, Singapore, 2017. He received the B.Eng. degree in Automation from Guangdong University of Technology, Guangzhou, China, in 2009 and the M.Sc. degree in Signal Processing from Nanyang Technological University, Singapore, in 2012, respectively. He worked as a Research Fellow with Future Mobility Research Lab, and Energy Research Institute @ NTU (ERI@N), Singapore. He is currently a Research Assistant Professor with the Department of Industrial Systems Engineering and Management, National University of Singapore, Singapore. His research interests include AI and combinatorial optimization problems.

Jie Zhang received the Ph.D. degree from Cheriton School of Computer Science, University of Waterloo, Canada, in 2009. He is an Associate Professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. During his Ph.D. study, he received the prestigious NSERC Alexander Graham Bell Canada Graduate Scholarship for top Ph.D. students across Canada. His research has been focused on the design of effective and robust intelligent software agents, through the modeling (trustworthiness, preferences) and simulation of different agents in a wide range of environments, using AI techniques (data mining, machine learning, and probabilistic reasoning) and multi-agent technologies.