Deep Reinforcement Learning for Combinatorial Optimization: Covering Salesman Problems

Kaiwen Li, Tao Zhang, Rui Wang, *Member, IEEE*, Yuheng Wang, Yi Han, and Ling Wang

***Abstract*—This article introduces a new deep learning approach to approximately solve the covering salesman problem (CSP). In this approach, given the city locations of a CSP as input, a deep neural network model is designed to directly output the solution. It is trained using the deep reinforcement learning without supervision. Specifically, in the model, we apply the multihead attention (MHA) to capture the structural patterns, and design a dynamic embedding to handle the dynamic patterns of the problem. Once the model is trained, it can generalize to various types of CSP tasks (different sizes and topologies) without the need of retraining. Through controlled experiments, the proposed approach shows desirable time complexity: it runs more than 20 times faster than the traditional heuristic solvers with a tiny gap of optimality. Moreover, it significantly outperforms the current state-of-the-art deep learning approaches for combinatorial optimization in the aspect of both training and inference. In comparison with traditional solvers, this approach is highly desirable for most of the challenging tasks in practice that are usually large scale and require quick decisions.**

***Index Terms*—Attention, covering salesman problem (CSP), deep learning, deep reinforcement learning (DRL).**

I. Introduction

THE TRAVELING salesman problem (TSP) is a frequently studied combinatorial optimization problem in the field of operation research. Given a set of cities with their spatial locations, the goal of TSP is to find a minimum length tour that visits each city once and returns back to the original city. In this work, we focus on the covering salesman problem (CSP), which is a generalization of the TSP. In the CSP, each city is given a predefined covering distance, within which all other cities are covered. The CSP seeks a minimum length tour over a subset of the given cities such that each city has to be either visited or has to be covered by at least one city on the tour. The CSP reduces to a TSP if the covering distance of each city is zero. Thus, the CSP is NP-hard and is more difficult to be solved than the TSP. In some practical scenarios, due to the limitation of fuel or manpower resources, it is hard to guarantee that each city can be traveled exactly once as assumed in the TSP. Hereby, the CSP arises in various and heterogeneous real-world applications, such as emergency management and disaster planning [1]–[3]. For example, in the routing problem of healthcare delivery teams [1], it is not necessary to visit each village, because the villages that are not visited are expected to go to their nearest stop for service.

Traditional approaches to solving such NP-hard combinatorial optimization problems mainly include three categories [4]: 1) exact algorithms; 2) approximate algorithms; and 3) heuristic algorithms. Exact algorithms can produce optimal solutions by enumeration or other techniques such as branch and bound [5], however, require forbidding computing time when tackling large instances. Approximate algorithms [6], in general, can obtain near-optimal solutions in polynomial time with theoretical optimality guarantees. However, such approximate algorithms might not exist for all types of optimization problems and the execution time can be still prohibitive if their time complexity is higher order polynomial. Heuristic methods, such as local search heuristics [7] and population-based heuristics [8], [9], are more favorable in practice since they usually run faster than the above two types of approaches. But they lack theoretical guarantees for the quality of the solutions. In addition, as iteration-based approaches, heuristic methods still suffer from the limitation of long computation time when tackling with large instances as a large number of iterations are required for population updating or heuristic searching. Moreover, esoteric domain knowledge and trial-and-error are usually required when designing such heuristic methods, leading to considerable designing effort and time.

As most of the challenging problems in real-world applications are large scale and are usually under the constraint of execution time, traditional algorithms suffer from specific limitations when applied to practical challenging tasks: forbidding computation time and the need to be revised or reexecuted whenever a change of the problem occurs. This can be impractical for large-scale tasks in real-world applications. Recent advances in deep learning have shown promising ability of solving NP-hard decision problems. A well-known example

Manuscript received 7 February 2021; revised 23 May 2021; accepted 4 August 2021. Date of publication 26 August 2021; date of current version 18 November 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 72071205, Grant 61873328, and Grant 61773390. This article was recommended by Associate Editor H. Ishibuchi. *(Corresponding authors: Tao Zhang; Rui Wang.)*

Kaiwen Li, Tao Zhang, and Rui Wang are with the College of System Engineering, National University of Defense Technology, Changsha 410073, China, and also with the Hunan Key Laboratory of Multienergy System Intelligent Interconnection Technology, HKL-MSI2T, Changsha 410073, China (e-mail: kaiwenli_nudt@foxmail.com; zhangtao@nudt.edu.cn; ruiwangnudt@gmail.com).

Yuheng Wang is with the Graduate College, National University of Defense Technology, Changsha 410073, China.

Yi Han is with the Science and Technology on Parallel and Distributed Processing Laboratory, College of Computer, National University of Defense Technology, Changsha 410073, China.

Ling Wang is with the Department of Automation, Tsinghua University, Beijing 100084, China.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCYB.2021.3103811.

Digital Object Identifier 10.1109/TCYB.2021.3103811

is the inspiring success of employing deep reinforcement learning (DRL) to solve the game Go [10], which is a complex discrete decision problem. In the context of the advances attained by the deep learning in solving various decision tasks, recently, some works have focused on using the deep learning to solve classical NP-hard combinatorial problems, including the TSP. They replace the carefully handcrafted heuristics by the policy parameterized by a deep neural network (DNN) and learn the policy from data. They provide a new paradigm for combinatorial optimization: the solution is directly output by a DNN, which is trained on a collection of instances from a certain type of problem. Such learning-based approaches have shown appealing advantages over traditional solvers in the aspect of computational complexity and the ability of scaling to unseen problems [11], [12].

In the literature, most of the studies are designing different types of heuristics to tackle CSP. These carefully designed heuristics can certainly improve the performance; however, they are problem specific and suffer from the aforementioned limitations. In this article, we propose a new deep learning approach to approximately solve CSP. The promising idea to leverage the deep learning for combinatorial optimization has been tested on TSP. However, as a generalization of the TSP, the CSP appears harder to be addressed due to its dynamic feature. Therefore, we propose a powerful DNN model based on multihead attention (MHA) and dynamic embedding that can effectively map from the problem input of the CSP to its solution. The model is trained using DRL in an unsupervised way. The proposed approach has empirically shown a handful of merits in the following aspects.

1) *Optimality:* On solving the CSP, the proposed approach significantly outperforms the recent state-of-the-art deep learning approaches for combinatorial optimization.
2) *Execution Time:* The presented approach runs more than 20 times faster than the traditional heuristic solvers with only a small optimality gap. It offers a desirable trade-off between the execution time and the optimality of solutions.
3) *Learning Ability:* Traditional solvers typically rely on a plethora of handcrafted expert heuristics, which can fail to exploit subtle statistical similarities between problem instances. The proposed method can learn data-driven policies from a category of similar problem instances that automatically account for such statistical relationships. Thereby, it can scale to unseen problem instances.
4) *Scalability:* Once trained, the model can be used to solve, not only one but a collection of CSP instances of different size and/or different city locations as long as they are from the same data generating function. It can scale to unseen instances with no need of retraining.
5) *Generalization:* The proposed approach can also generalize to different CSP tasks, for example, CSPs where each city has a fixed covering radius within which all cities are covered, or CSPs where each city can cover a fixed number of cities. The model that is only trained on one type of CSP task; however, it can solve different types of CSP tasks without the need to retrain the model.

II. Related Work

A. Covering Salesman Problems

The CSP is first formulated in 1989 by Current and Schilling [1]. They presented a straightforward heuristic approach to solve it. It can be divided into two parts. First, a minimum number of cities that can cover all the cities are selected, namely, a set covering problem (SCP). The second phase can be regarded as a TSP: the shortest tour is found on the above selected cities. Since usually more than one solution can be found for the SCP, the TSP solver is applied on all the found solutions and the TSP solution with the minimum length is output as the CSP solution.

From then on, a number of good heuristics are designed to solve the CSP effectively. Golden *et al.* [13] proposed two local search algorithms (LS1 and LS2) to solve CSP. LS1 and LS2 have been widely used as benchmark approaches in the literature. As local search methods, they all start from a random feasible solution and improve it by making use of different operators, such as destroy, repair, and permutation. LS1 first removes a fixed number nodes from the current solution according to a deletion probability. New nodes are then inserted into the current tour one by one according to a insertion probability until the solution is feasible again. These probabilities are determined by the decrease or increase of the tour length when deleting or adding that node into the tour. In contrast, LS2 removes one node a time from the current solution and then insert the nearest nodes from the removed node into the solution. Lin–Kernighan procedure and 2-Opt procedure are also used to improve the solution.

Salari and Naji-Azimi [14] developed an integer linear programming (ILP)-based heuristic method to solve the CSP. Its main difference from the method of [13] is its idea of applying ILP to improve the solution. This approach first applies the similar destroy and repair operations to decrease the tour length by removing some nodes from the current solution and inserting new nodes back to make a feasible solution. The solution is further improved by solving an ILP model with the objective of minimizing the tour length. Other local search approaches also apply the similar destroy and repair operations to solve the CSP. For example, Shaelaie *et al.* [15] proposed a variable neighborhood search method and Venkatesh *et al.* [16] proposed a multistart iterated local search algorithm.

In addition to the local search approaches, various population-based heuristic approaches have been proposed. Salari *et al.* [17] combined the ant colony optimization (ACO) algorithm and dynamic programming to tackle the CSP. It also incorporates various local search procedures, such as removal, insertion, and the 3-OPT search. The experiments show this approach to be superior on large size instances. Tripathy *et al.* [18] presented a genetic algorithm (GA) with new designed crossover operators for solving the CSP. However, its performance fails to defeat LS1 and LS2. Different types of GAs [15], [19] have also been developed to solve the CSP. In addition, Pandiri and Singh [20] proposed an artificial bee colony algorithm for this problem.

In terms of the performance, the above heuristic algorithms achieve a same level of optimality to the earlier

proposed LS1/LS2 on most instances. Some of the heuristics may outperform LS1/LS2 on several instances. For years, various heuristics have been designed to solve CSP with significant specialized knowledge and trial-and-error efforts. However, there was no significant breakthrough, and currently no machine learning methods have been proposed for the CSP to the best of our knowledge.

B. Deep Learning Approaches for Combinatorial Optimization

Recent five years have seen a surge in the applications of deep learning for combinatorial optimization. Deep learning approaches for combinatorial optimization are usually based on the *end-to-end* learning mode, that is, using a DNN to directly output the optimal solution. Current state-of-the-art approaches mainly use *sequence-to-sequence* networks [22] and graph neural networks (GNNs) [23] for combinatorial optimization. Representative works are reviewed as follows.

Vinyals *et al.* [11] first proposed to use a sequence-to-sequence model for combinatorial optimization, also known as the pointer network (PN). It uses the attention mechanism to output a permutation of an input sequence. The model is trained offline using pairs of TSP instances and their (near) optimal tours in a supervised fashion. It successfully solves the small size TSP and reinvigorates this line of work that applies deep learning for combinatorial optimization.

Learning from supervised labels might be inapplicable because it is expensive to construct high-quality labeled data and it prohibits the learned model from performing better than the training examples. In this context, Bello *et al.* [24] proposed to use an actor–critic [25] DRL algorithm to train the PN in an unsupervised manner. It takes each TSP instance as a training sample and uses the tour length of the solution obtained by the current policy as the reward, which is used to update the policy parameters via the policy gradient formula. This approach performs competitively to [11] on small instances and can solve larger TSP instances (up to 100 cities). Nazari *et al.* [26] replaced the LSTM [27] encoder of the PN by a simple node embedding. It can save up to 60% training time for the TSP while maintaining a similar performance. This model can also solve the vehicle routing problem (VRP) by incorporating additional dynamic elements.

Khalil *et al.* [28] considered the graph structure of the combinatorial optimization problem by adopting a *structure2vec* GNN model. It consecutively inserts a node into the current partial solution according to the node scores parameterized by *structure2vec*. The model is trained using the DQN method [29] and is applied on the minimum vertex cover and maximum cut problems other than the TSP. Mittal *et al.* [30] replaced the *structure2vec* of [28] by a graph convolutional network (GCN) and followed the same greedy algorithm. It performs better than [28] on large graphs.

Nowak *et al.* [31] also used a GNN to model the problem. But it outputs an adjacency matrix instead, which is then converted into a feasible solution by beam searching. The model is trained with supervision. As this method constructs solutions in a nonautoregressive manner, it performs worse than the above autoregressive approaches for the TSP. Joshi *et al.* [32] followed the same paradigm to [31] but used a more effective GCN to encode the problem instance. The experiments show its superior performance on the TSP when using beam search and shortest tour heuristic to construct the solutions, however, requiring much longer execution time. Moreover, Li *et al.* [33] used a GCN to output the probability map that represents the likelihood of each node belonging to the optimal solution. Then, a guided tree search is used to construct the solution according to the probability map of all nodes.

Inspired by the transformer [34], which is the state-of-the-art model in the field of sequence-to-sequence learning, Croes [12] and Deudon *et al.* [35] applied the MHA mechanism of the transformer to tackle the combinatorial optimization challenge. They build the TSP solutions autoregressively using the attention similar to the PN. Deudon *et al.* [35] demonstrated that incorporating a 2-OPT local search [36] can improve the performance. Kool *et al.* [12] designed a more effective decoder and proposed to train the model using a new reinforcement learning method with a greedy rollout baseline. This approach achieves the state-of-the-art performance among the concurrent deep learning approaches for the TSP. Moreover, it can scale to a variety of practical problems, such as the VRP, orienteering problem, etc. Furthermore, Ma *et al.* [37] proposed a graph PN that combines the GNN and the PN to tackle the TSP.

In the case where multiobjective combinatorial optimization problems are mostly studied based on the heuristic methods [38]–[40], Li *et al.* [41] adopted a deep learning approach to solve the multiobjective TSP. This approach outperforms traditional multiobjective optimization solvers in terms of execution time and solution quality. In addition, other combinatorial optimization problems tackled by similar deep learning models include maximal independent set [42], graph coloring [43], Boolean satisfiability [43], etc. Recently, Joshi *et al.* [44] explored the impact of different training paradigms, and revealed favorable properties of reinforcement learning over supervised learning. Sato *et al.* [45] investigated the approximation ratios of GNNs for combinatorial problems. The readers are referred to [4] and [46] for a more extensive discussion of neural combinatorial optimization.

The remainder of this article is organized as follows. Section III formally defines the CSP and introduces the fundamental theory of how to apply deep learning to solve the combinatorial optimization problem. Our proposed attention-dynamic model for CSP is described in Section IV. Section V outlines the training method for optimizing the model parameters. The experiment setup is specified in Section VI, and computational results are presented in Section VII. Finally, Section VIII gives some concluding remarks and future perspectives.

III. Preliminaries

A. Problem Definition

In the CSP, we are given a complete graph $G = (V, E)$ where $N = \{1, 2, \ldots, n\}$ is the node set that represents the cities. $E = (\{i, j\} : i, j \in N, i < j)$ is the edge set, and c_{ij} represents the cost of edge $\{i, j\}$, which is usually defined as

the shortest distance between node i and j. Each node i can cover a subset of nodes S_i. In the CSP, we are expected to find a Hamiltonian tour over a subset of nodes V such that cities that are not on the tour must be covered by at least one city on the tour. The objective is to minimize the total tour length.

In this article, we focus on the Euclidean CSP: the cost of edge $\{i, j\}$ is defined by the Euclidean distance between node i and j given their 2-D geographical locations.

B. Attention Theory for Combinatorial Optimization

For ease of understanding our model in this article, we first introduce the fundamental theory of how to use deep learning techniques to solve the combinatorial optimization problem, the TSP specifically.

Node Embedding: The initial *node embedding* of node i is a mapping of the 2-D input of node i, that is, 2-D coordinates of its location, to a d_h-dimensional vector ($d_h = 128$ in this work).

PN: The model architecture of PN is shown in Fig. 1. Its basic structure is the encoder–decoder. The encoder is used to encode the node embeddings to a *Vector*, while the decoder is to decode the *Vector* into an output sequence. Either the encoder or decoder is a recurrent neural network (RNN). While using the encoder RNN to encode the inputs, the hidden state e_i is produced for each city i. Here, e_i can be interpreted as the feature vector of city i. The final hidden state of the encoder RNN is the *Vector*.

As indicated in Fig. 1, the *Vector* is used as input to the decoder RNN, and the decoding happens sequentially. The *Vector* also serves as the initial hidden state d_0, which is used to select the first city to visit. The next hidden state d_1 is obtained by inputting the node embedding of the selected city to the encoder RNN. Specifically, at decoding step t, the probability of city selection is calculated by the encoder state d_{t-1} and the feature vector e_i of each city. The RNN reads the node embedding of the selection and outputs the current encoder state d_t, which is then used in the next decoding step. The calculation of the probability is, namely, the attention method and is computed as follows:

$$\begin{aligned} cu_i^t &= v^T \tanh(W_1 e_i + W_2 d_t) \quad i \in (1, \ldots, n) \\ P(y_{t+1}|y_1, \ldots, y_t, X) &= \text{softmax}\left(u^t\right) \end{aligned} \tag{1}$$

where v, W_1, and W_2 are learnable parameters. The one with the largest probability can be selected as the city to be visited at step t.

For example, as shown in Fig. 1, to select the first city at decoding step $t = 1$, u_1^1, u_2^1, and u_3^1 are calculated by (d_0, e_1), (d_0, e_2), and (d_0, e_3) according to (1). The one with the largest u^1, that is, u_2^1, is selected as the first city, which is city 2. By inputting the node embedding of city 2 into the decoder RNN, we can get d_1. The next city is selected according to the probabilities calculated by applying the attention formula on (d_1, e_1), (d_1, e_2), and (d_1, e_3). Then, the procedure loops till all the cities are selected. After training the parameters v, W_1, and W_2 offline, the PN obtains the ability to output the desired tour of the cities.

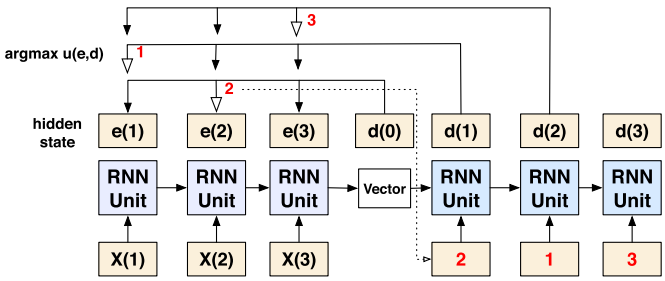


Fig. 1. Take a 3-city TSP instance as an example to demonstrate the pointer network. Inputs are the embeddings of the city locations. The cities are selected step by step. At each step t, the city with the largest u^t is selected. u_i^t for each city i is calculated according to (1).

Self-Attention: In the above PN, RNN is used to get the feature vector e_i of each city i in the encoder. Self-attention achieves the same functionality. It is the basic component of the transformer [34], which is the state-of-the-art model in the field of sequence-to-sequence learning such as machine translation. By applying self-attention in the encoder, the obtained attention value of each input i stores not only the information of itself but also its relations with other inputs. Thus, self-attention can be more effective in capturing the structural patterns of the problem.

Recalling the attenton mechanism used in (1), d_t in the tth city selecting step can be seen as a *query*. It contains the information of already visited cities. The representation e_i of each city i can be seen as a *key*. The ith city is given more *attention* if its *key* is more compatible with the *query*. In (1), the *attention* value is interpreted as the probability of being selected.

Self-attention works by calculating the attention value of each component in a sequence to all other components. The obtained attention value of the ith component serves as its representation. Therefore, the processed embeddings contain more information compared to its original sequence. Different from the attention calculation method in (1), the *scaled dot-product attention* is used in transformer's self-attention: the compatibility of query $\mathbf{q}_t$ and key $\mathbf{k}_i$ is simply calculated as their multiply $\mathbf{q}_t^T \mathbf{k}_i$ and then scaled by a scaling factor. The self-attention mechanism is introduced as follows.

The basic attention is realized by *query* and the *key-value* pair, that is, more attention would be given to *value* if its *key* is more compatible with *query*. For self-attention in the encoder, *query*, *key*, and *value* are all linear projections of the input node embeddings

$$\mathbf{Q} = W^Q \mathbf{X}, \quad \mathbf{K} = W^K \mathbf{X}, \quad \mathbf{V} = W^V \mathbf{X}. \tag{2}$$

That is, the *query*, *key*, and *value* of the ith node is

$$\mathbf{q}_i = W^Q \mathbf{X}_i, \quad \mathbf{k}_i = W^K \mathbf{X}_i, \quad \mathbf{v}_i = W^V \mathbf{X}_i. \tag{3}$$

The process of calculating the attention value of the ith node is depicted in Fig. 2. First, the relations of the ith node with other nodes are computed by a compatibility function of the *query* of the ith node with the *keys* of all other nodes: $\mathbf{q}_i \mathbf{K}^T$. The obtained compatibility serves as weights and is normalized by the softmax function. The attention value is finally computed as a weighted sum of the *values* $\mathbf{V}$, where

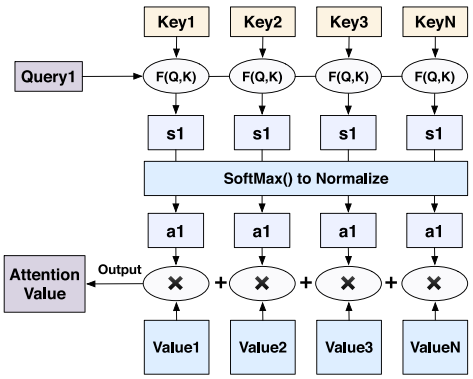


Fig. 2. Process of calculating the attention value given the *query* and *key-value* pair.

the weights are $\mathbf{q}_i\mathbf{K}^T$. Thus, the attention value of the ith node is $\mathbf{q}_i\mathbf{K}^T\mathbf{V}$, which has the same dimension (d_h) with its initial node embedding $\mathbf{X}_i$. As a result of the self-attention process, the computed attention value of the ith node stores not only its own knowledge but also its relations with other nodes.

In practice, we compute the attention values of all nodes simultaneously by matrix multiplying

$$\text{Attention } (Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4}$$

where $\sqrt{d_k}$ is the scaling factor and d_k is the dimension of the *key*.

MHA: As pointed out in [34], it is beneficial to carry out the attention process multiple times and obtain M attention values. Then, the M attention values are linearly projected into a final attention value. Hence, various features can be extracted using the MHA process.

Overall, structural patterns of the problem can be extracted effectively by using the MHA in the encoder.

IV. Proposed Attention-Dynamic Model

We still follow the *encoder–decoder* architecture to model the CSP. The encoder is used to construct the *key* of the model that consists of two parts: 1) the static embedding obtained by the MHA that contains the structural patterns of the problem and 2) the dynamic embedding obtained by a designed guidance vector that describes the dynamic patterns of the problem. The decoder is designed with an RNN and an additional attention operator.

The model is formally described as follows. Taking a CSP instance s with n cities as an example, the feature $\mathbf{x}_i$ of city i is defined by its location, that is, its x-coordinate and y-coordinate. A solution $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_k, k \le n)$ is a permutation of a subset of the cities. All the cities should be visited or covered by traveling along the tour $\boldsymbol{\pi}$. With this formulation of CSP, the attention-dynamic model is designed in this article. The model takes as input city locations of the instance s, and outputs the sequence $\boldsymbol{\pi}$. More formally, the model defines a policy $p(\boldsymbol{\pi}|s)$ for selecting $\boldsymbol{\pi}$ given s

$$p_\theta(\boldsymbol{\pi}|s) = \prod_{t=1}^{k} p_{\boldsymbol{\theta}}(\pi_t|s, \boldsymbol{\pi}_{1\sim t-1}), k \le n. \tag{5}$$

The encoder encodes the original features $\mathbf{x}$, and produces the final embeddings (representations) of the cities. The decoder takes as input the encoder embeddings, summarizing the information of previously selected cities $\boldsymbol{\pi}_{1\sim t-1}$, and then outputs π_t, one city at a time. $\boldsymbol{\theta}$ represents the model parameters, such as W^Q, W^K, and W^V. With an optimal set of $\boldsymbol{\theta}^*$, the model has the ability of outputting the optimal tour $\boldsymbol{\pi}^*$ given a problem instance s.

A. Encoder

The purpose of the encoder is to produce a representation for each city, that is, produce the *key* of each city. Our *key* consists of two parts: 1) static embeddings and 2) dynamic embeddings.

Construction of Static Embeddings: The static embedding demonstrates the static feature of a city, that is, its location. It is constructed similar to the TSP task in [12] and [35]. We employ the introduced MHA to produce the static embeddings, as shown in Fig. 3. Its basic element is the *self-attention*, which can produce the representation for each city i that stores not only its own feature but also its relations with other cities. As depicted in Fig. 3, in the MHA, the self-attention is conducted h times and the obtained h embeddings are projected into a final embedding.

The architecture of computing the static embeddings consists of N sequentially connected *attention layers* that all have the same structure. That is, the embeddings are processed N times sequentially. Each attention layer has two sublayers: the first is an MHA layer, and the second is a fully connected feedforward layer. In addition, each sublayer is added with a residual connection [47] and a layer normalization. Thus, the output of each sublayer becomes $\text{Norm}(x + \text{Sublayer}(x))$.

Overall, computing static embeddings needs two steps.

1) We first map the d_x-dimensional city locations ($d_x = 2$) to the d_h-dimensional node embeddings ($d_h = 128$)

$$\mathbf{h}_i^{(0)} = W^{\mathbf{x}}\mathbf{x}_i + \mathbf{b}^{\mathbf{x}}. \tag{6}$$

$W^{\mathbf{x}}$ and $\mathbf{b}^{\mathbf{x}}$ are learnable parameters. Assuming there are n cities, thus the shape of the initial node embeddings is $n \times d_h$.

2) As shown in Fig. 3, the initial node embeddings $h^{(0)}$ are processed and updated through N attention layers. The output embeddings of the ℓth layer are computed as

$$\begin{aligned}\mathbf{h}_{tmp} &= \text{Norm}^\ell\left(\mathbf{h}^{(\ell-1)} + \text{MHA}^\ell\left(\mathbf{h}^{(\ell-1)}\right)\right)\\ \mathbf{h}^{(\ell)} &= \text{Norm}^\ell\left(\mathbf{h}_{tmp} + \text{FF}^\ell\left(\mathbf{h}_{tmp}\right)\right).\end{aligned} \tag{7}$$

The output embedding $h^{(N)}$ of the Nth layer is the final static embedding, where $h_i^{(N)}$ is the d_h-dimensional static feature vector of city i.

Construction of Dynamic Embeddings: The static embeddings $h^{(N)}$ can be directly taken as the *key* for the TSP [12]. However, the states of the cities in a CSP are not static during the decoding step. For example, once a city is covered, its possibility of being visited might be reduced. The possibility can be rather reduced if it is covered more times. Thus, it is

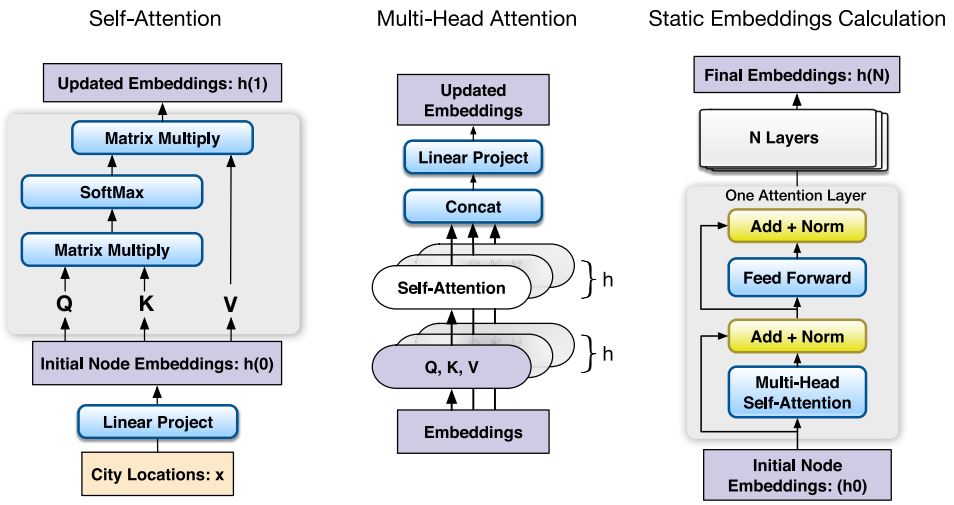


Fig. 3. This figure gives the process of calculating the static embeddings in our model. It consists of N layers of MHA modules, which is formed by the self-attention mechanism.

vital to introduce the concept of *covering* into the construction of *key* for CSP optimization. Aiming at this, we define a guidance vector g_i for each city i that indicates its state of covering, and g_i is dynamically updated at each decoding step of t.

Assuming that city π_t is selected to be visited at decoding step t, $C(\pi_t)$ is defined as the set of cities that are covered by π_t. The number of nodes in $C(\pi_t)$ is counted as $N_{C(\pi_t)}$. Cities in $C(\pi_t)$ are then sorted according to their distances to π_t. Thus, each city in $C(\pi_t)$ has a sorted index c_i ranging from 1 to $N_{C(\pi_t)}$, where $c_i = 1$ indicates that city i is nearest to π_t, and vice versa.

First, g_i is initialized to 1 for all cities

$$g_i = 1, i = 1, \ldots, n. \tag{8}$$

At each decoding step t, g_i is updated dynamically

$$g_i = g_i \times \frac{c_i}{N_{C(\pi_t)}}, i \in C(\pi_t), i = 1, \ldots, n. \tag{9}$$

According to (9), the dynamic state of city i is represented by a value g_i. As shown in Fig. 4, once a city A is visited, the g_i values of A's surrounding cities, that is, cities that are covered by A, are decreased according to their distance to A. The nearest city to A is assigned with the smallest value. g_i is further reduced if city i is covered again by other cities.

Hence, the obtained guidance vector contains the covering information for each city: the state that it is covered or not and its distance to the one who covers it. It can thus guide the selection of the next city, for example, the one that is closer to the already visited cities may be less possibly to be selected.

Note that we do not explicitly reduce the selection probability of city i according its g_i value. It is because that a covered city can also be a good option in some occasions. Instead, we let the model itself to learn how to utilize g_i to adjust the probability. To achieve this, g_i is projected to a d_h-dimensional embedding $G_i = g_i W_G$ by learnable weights W_G.

As G_i dynamically changes during the decoding, G_i is called the dynamic embedding in this study. While the static

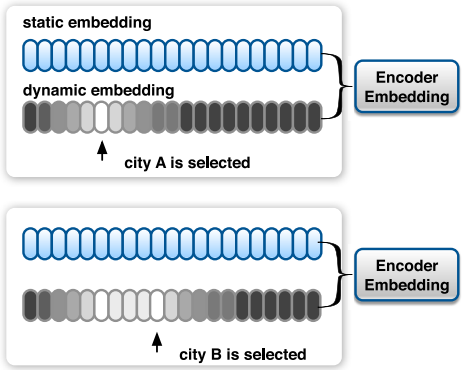


Fig. 4. This figure shows the changes of g_i values if a city is visited. The darker the color, the larger the g_i value. g_i is projected to the dynamic embedding, and the encoder embedding is constructed by combining the static and dynamic embedding.

embedding $h^{(N)}$ indicates the static states of the cities, that is, their spatial locations, the dynamic embedding G indicates their dynamic states of *covering*. Then, we construct the *key* according to $h^{(N)}$ and G

$$\mathrm{k}_i = W^K h_i^{(N)} G_i, i = 1, \ldots, n \tag{10}$$

where $\mathbf{k}_i$ represents the *key* for city i and W^K are the learnable weights with size $(d_h \times d_h)$. In this way, the model can capture both of the static and dynamic patterns of the problem and, thus, learns better policies.

B. Decoder

The decoder is composed of two parts: 1) RNN, whose output d_t is used to compute the *query* q_t at decoding step t and 2) encoder–decoder attention, which is used to calculate the probability of city selections according to *query* q_t and *key*. Intuitively, the city with the largest probability is selected as π_t at decoding step t. The probability is calculated through the encoder–decoder attention based on the *query* q_t from the RNN and the *key* from the encoder. That is, among all key_i,

$i = 1, \ldots, n$, the one that most matches q_t is selected as the next city to be visited. This part illustrates how to build the *query* in the model.

Construction of Query: To select the next city to be visited, it is natural to take the previously visited cities as the *query*. As RNN is capable of handling such sequential information, in this study, GRU [48], which is a variant of RNN, is employed to construct the *query* in the decoder. We set the mean of the encoder embeddings $\overline{\mathbf{h}} = 1/n \sum_{i=1}^{n} \mathbf{h}_i^{(N)}$ as the initial hidden state of the decoder RNN, that is, $d_0 = \overline{\mathbf{h}}$. Meanwhile, we use the learned parameter $\mathbf{v}_1$ as the first input to the decoder RNN at step $t = 1$. In the following steps $t > 1$, the input to the RNN is the node embedding $\mathbf{h}_{\pi_{t-1}}^{(N)}$ of the previous selection π_{t-1}

$$o_t, d_t = f_{\text{GRU}}(I_t, d_{t-1}), d_0 = \overline{\mathbf{h}} \tag{11}$$

$$I_t = \begin{cases} \mathbf{v}_1, & t = 1 \\ \mathbf{h}_{\pi_{t-1}}^{(N)}, & t > 1 \end{cases} \tag{12}$$

where d_t is the hidden state of the decoder RNN at decoding step t, and it stores the information of previously selected cities $\pi_{1\sim t-1}$. As the selection of π_t largely depends on $\pi_{1\sim t-1}$, d_t can be employed as the *query* naturally.

However, $\pi_{1\sim t-1}$ is not the only factor that determines π_t. To construct the *query*, the relations between the already visited cites and the rest of cities should also be considered. For example, the city that is closer to the last visited city might have more opportunity to be visited next. Thus, we further process d_t using the attention mechanism as follows:

$$\mathrm{q}_t = \text{Attention}(d_t, K_1, V_1) = \text{softmax}\left(\frac{d_t K_1^T}{\sqrt{d_k}}\right) V_1 \tag{13}$$

where K_1 and V_1 are linear projections of the static embeddings $h^{(N)}$ from the encoder. Thus, the *query* q_t for selecting π_t is defined as the attention value of d_t and the key-value pairs K_1 and V_1. By adding the extra attention operation to d_t, the obtained *query* contains richer information: the compatibility of previously selected cities (represented by d_t) with all other cities [represented by $h^{(N)}$]. This operation further extracts the location relationships between the cluster of already visited cities and the rest cities, which can aid the next decision.

C. Compute Probability by Attention

At the decoding step t, we compute the selection probability of all the cities $i = 1, \ldots, n$ based on the *query key* (q_t, k_i) by the scaled dot-product attention

$$u_i^t = \frac{\mathbf{q}_t^T \mathbf{k}_i}{\sqrt{d_k}} \tag{14}$$

where $\sqrt{d_k}$ is the scaling factor and $d_k = d_h/M$. M is the number of heads in the MHA. In addition, we mask the cities that are already visited to make sure they will not be selected again

$$u_i^t = \begin{cases} -\infty, & \text{if } j \in \pi_{1\sim t-1} \\ \frac{\mathbf{q}_t^T \mathbf{k}_i}{\sqrt{d_k}}, & \text{otherwise.} \end{cases} \tag{15}$$

The probability will be zero after applying the softmax operator to the already visited cities with $u_i^t = -\infty$. The final probability is computed by scaling u_i^t using the softmax operator

$$p_i = \frac{e^{u_i^t}}{\sum_{j=1}^{n} e^{u_j^t}}, i = 1, \ldots, n. \tag{16}$$

Moreover, the cities that are covered by at least one city on the tour are masked as well; however, their probabilities of being selected are not set to zero. We let the model itself learn to how to adjust the probability.

Assuming that the greedy strategy is used, the city with the largest probability p_i is selected to be visited at decoding step t. It is noted that the city is randomly sampled from the probability distribution while training, and is greedily selected according to the probability at the inference stage.

The model is, therefore, composed of the above encoder, decoder, and attention. The key contribution is the design of the dynamic embedding, which can effectively understand the covering information of the CSP. Moreover, the applied MHA mechanism can extract richer location features of the problem. The decoder is designed with an additional attention operation to aid the decoding decision. Based on the designed neural architecture, the encoder takes as input the features of the CSP instance and outputs the encoder embeddings. Decoding happens sequentially from $t = 0$ and stops until all the cities are visited or covered. The selected cities form the final solution $(\pi_0, \pi_1, \ldots)$, that is, permutation of the cities.

V. Train With REINFORCE Algorithm

Given a CSP instance s, Section IV defines the model parameterized by parameters $\boldsymbol{\theta}$ that can produce the probability distribution $p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)$ by (15) and (16). The solution $\boldsymbol{\pi}|s$ can be obtained by sampling from $p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)$. Since the goal is to minimize the total length $L(\pi)$ of the cyclic tour, it is suitable to train the model with the REINFORCE algorithm [49], which uses Monte Carlo rollout to compute the rewards, that is, play through the entire episode to find a complete cyclic tour and the total tour length is computed as the reward ($-L(\pi)$). Then, the agent can learn to improve itself by the *state–action–reward* tuple.

Formally, given the actions $\boldsymbol{\pi}|s$ and the corresponding reward/loss $L(\pi)$, parameters $\boldsymbol{\theta}$ of the model can be updated by gradient descent using the REINFORCE algorithm [49]

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) &= \mathbf{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)}\left[\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s) L(\boldsymbol{\pi})\right] \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}). \end{aligned} \tag{17}$$

Since a number of city selection actions are sampled from the probability distribution and the total length is computed as the rewards, the recorded rewards of different episodes have high variance due to the uncertainty of the sampling. The variance provides conflicting descent direction for the model to learn and hurt the speed of convergence. To reduce the variance, it is common to introduce the baseline $b(s)$ to rewrite the policy gradient in (17)

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \mathbf{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)}\left[\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)(L(\boldsymbol{\pi}) - b(s))\right]. \tag{18}$$

Loosely speaking, $b(s)$ serves as the average performance. A policy is encouraged if its action performs better than the

Algorithm 1 Training Algorithm Using REINFORCE

Input: training set $\mathcal{X}$, batch size B, number of epochs, N_{epoch}, number of steps per epoch N_{steps}

Initialize model parameters of the current policy and baseline policy $\boldsymbol{\theta}, \boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
for $epoch \leftarrow 1 : N_{epoch}$ **do**
  for $step \leftarrow 1 : N_{step}$ **do**
    $s_i \leftarrow \text{RandomInstance}(\mathcal{X})$ for $i \in \{1, \ldots, B\}$
    $\boldsymbol{\pi}_i \leftarrow p_{\boldsymbol{\theta}}(s_i)$. Run the policy by sampling
    $\boldsymbol{\pi}_i^* \leftarrow p_{\boldsymbol{\theta}^*}(s_i)$. Run the policy greedily
    $\nabla \mathcal{L} \leftarrow \sum_{i=1}^{B} \big(L(\boldsymbol{\pi}_i) - L\big(\boldsymbol{\pi}_i^*\big)\big) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{\pi}_i)$
    Update $\boldsymbol{\theta}$ by Adam according to $\nabla \mathcal{L}$
  end for
  if $p_{\boldsymbol{\theta}}$ performs better than $p_{\boldsymbol{\theta}^*}$ **then**
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
  end if
end for

average. A popular method is to train an additional neural network, that is, the critic network, to approximate $b(s)$. Its parameters are learned via pairs of $(s, L(\boldsymbol{\pi}))$.

However, as reported in [12], training a policy network and a critic network simultaneously is hard to converge. Therefore, we follow the greedy rollout baseline as introduced in [12], which is proved to be superior than the critic baseline in the traditional actor–critic training algorithm.

Here, no additional neural network is required to approximate $b(s)$. Instead, given a CSP instance s, $b(s)$ is computed as the length of the tour from a deterministic greedy rollout of the baseline policy $p_{\boldsymbol{\theta}^*}$. The greedy rollout of a policy means that a solution is built by running the policy greedily, that is, at each step, the city with the largest output probability is selected. The baseline policy is the best model so far during training. At the end of each training epoch, once the trained model is better than the baseline policy, we replace the baseline policy with the current model.

Algorithm 1 outlines the REINFORCE training algorithm with the greedy rollout baseline and Adam optimizer [50]. In this way, if a sampled solution $\boldsymbol{\pi}$ is better than the greedy rollout of the best model ever, $(L(\boldsymbol{\pi}) - b(s))$ is negative and, thus, reinforcing such actions, and vice versa. Thus, the model can effectively learn to solve the problem, that is, minimizing the total tour length.

VI. Experiment Setup

We investigate the performance of the proposed approach on randomly generated 20-, 50-, 100-, 200-, and 300-CSP tasks and instances generated from the TSPLIB.

Baseline Algorithms: The presented approach (attention-dynamic) is compared to both of the recent state-of-the-art deep learning techniques as well as classical nonlearned heuristic methods. The most popular deep learning methods on solving combinatorial optimization problems in the past few years are included in the comparisons.

1) PN [11].
2) PN with dynamic embeddings [26] (PN-dynamic).
3) Attention model [12] (Attention).

TABLE I
Hyperparameters Configurations

HyperParameters	Value	HyperParameters	Value
Batch size	256	Hidden dimension	128
No. of epoch	50	No. of heads	8
No. of instances per epoch	320000	No. of Encoder layers	3
Optimizer	Adam	Learning rate	1e-4

The PN model [11] is the first deep learning model that can effectively tackle combinatorial optimization problems. It is a widely used baseline approach. The PN-dynamic model [26] aims at solving the VRP problem whose state changes during the decision process that has the similar dynamic feature with CSP. An Attention model [12] is the current state-of-the-art deep learning method for solving various combinatorial optimization problems. Thus, the above models are included as competitors. They can be directly used to solve the CSP by simply changing their mask function, that is, the cities that are covered by at least one city on the tour are still masked.

Before the emergence of deep learning techniques, heuristic methods are the main approaches to solve CSP. Therefore, in addition to deep learning methods, we also include non-learned heuristic methods for comparisons. LS1 and LS2 [13] are commonly used benchmark algorithms for solving CSP from the literature [13]–[15], [17]. Since all the deep learning models run on Python platform and no source code is found for LS1/LS2, we implemented LS1/LS2 in Python to make fair comparisons. The code is written strictly according to [13] and is publicly available.[1]

Hyperparameters Configurations: Hyperparameters used for training our model are shown in Table I. Across all experiments of our model and the compared neural combinatorial baselines, we use minibatches of 256 instances, 128-dimensional node embeddings, and encoder–decoder with 128 hidden units. Models are trained using the Adam optimizer [50] with a fixed learning rate 10^{-4}. In the training phase, 320 000 CSP instances are generated and used in training for 50 epochs. The training CSP instances are randomly generated: city locations are randomly chosen from a uniform distribution that ranges from $[0, 1] \times [0, 1]$; the number of nearest neighbor cities that can be covered by each city (NC) is set as 7. It is worth noting that NC is set to a fixed number upon training the model; however, it can be set as an arbitrary number when testing. This validates the generalization ability of the model and will be further discussed in Section VII-C. We use the same set of hyperparameters and the same training dataset with fixed random seed for training other neural combinatorial baseline models. As for LS1 and LS2, we follow the recommended hyperparameters from [13] for different problem dimensions.

Implementations: All experiments are implemented in Python and conducted on the same machine with one GTX 2080Ti GPU and Intel 64GB 16-Core i7-9800X CPU. Our code is an open source[2] to reproduce the experimental results and to facilitate future studies.

[1] https://github.com/kevin031060/CSP_Attention/tree/master/LS
[2] https://github.com/kevin031060/CSP_Attention

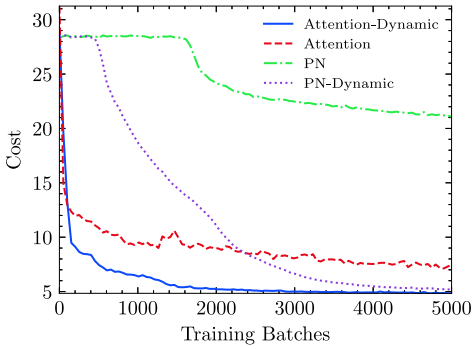


Fig. 5. Held-out validation set cost as a function of the number of training batches for the compared models.

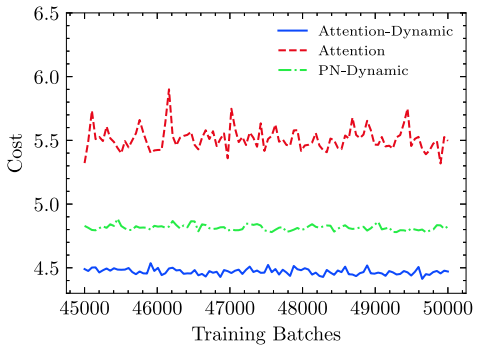


Fig. 6. Final costs of PN-dynamic, Attention, and Attention-dynamic at the end of the training phase.

Evaluation Procedure: We evaluate the performance on held-out test set of randomly generated 100 instances, and take the average predicted tour length as the performance indicator. Since the compared LS1 and LS2 require long execution time, we only generate 100 instances for testing. As pointed out in [35], a posterior local search can effectively improve the quality of solutions obtained by the neural combinatorial model. Thus, we conduct a simple local search to further process the obtained tour of our model. The running time of our method is the sum of the inference time of the neural network model and the execution time of the local search. Moreover, the performance of the proposed model on instances generated from the TSPLIB is also evaluated.

VII. Results and Discussion

The learning, approximation, and generalization ability of the presented approach are demonstrated in this section.

A. Comparisons Against Deep Learning Baselines on Training

We first evaluate the learning ability of our method in the training phase against the recent state-of-the-art deep learning baselines: PN, PN-dynamic, and Attention model.

Fig. 5 compares the performances of the compared approaches during training. The average predicted tour length of the held-out validation set is taken as the cost to evaluate the models in the training phase. As the lines may be too close, Fig. 6 shows the final cost of the compared models at the end of the training phase.

We observe that our method clearly outperforms the compared baselines in terms of both the convergence speed and the final cost. PN and Attention model fail to converge to the optimum as they do not take into account the dynamic information of the covering state of the cities. Although the PN-dynamic model considers the dynamic features, it is defeated by our model since the MHA mechanism that we used can effectively extract the feature of the CSP task; thus, it helps converge significantly faster and lead to a lower cost than the PN-dynamic model. Our approach appears to be more stable and sample efficient in comparison with the baselines.

TABLE II
Comparisons of Training Time for 100 Batches (in Seconds)

Method	CSP50	CSP100
PN	85.6	176.2
PN-dynamic	107.5	237.8
AM	26.3	45.9
AM-dynamic	32.6	52.6

In addition, Table II outlines the training time of the compared models. Despite the low performance of the PN-based models, they appear to be computational inefficient during training. Our method can save up to 75% of the training time while achieving a lower cost than the PN-dynamic model. Overall, training our models for 50 epochs on CSP50, CSP100, and CSP200 instances requires 3.6, 5.8, and 11.3 h with a single RTX2080Ti GPU.

B. Model Performance on Test Set

To evaluate the model performances, both the small-scale CSP tasks (CSP20, CSP50, and CSP100) and the larger scale tasks (CSP150, CSP200, and CSP300) are used for testing. We mainly focus on learning from small-scale training instances and measuring the performance to larger and unseen test instances: the model trained on CSP50 instances is used to solve randomly generated CSP20 and CSP50 test problems; the model trained on CSP100 instances is used to solve CSP100 and CSP150 test problems; and the model trained on CSP200 instances is used to solve CSP200 and CSP300 test problems. We empirically find that the performances deteriorate significantly for all deep learning methods when using model trained on CSP20 to solve CSP300 instances. Therefore, to fairly evaluate the model performance, we do not use such experimental setup with a large gap of the instance size between training and testing. We use the average predicted tour length as the cost to evaluate the model. The optimality gap of the approaches w.r.t. the best solver is also used as the performance metric.

Tables III and IV present the performances of the compared approaches on small-scale CSP tasks (CSP20, CSP50, and CSP100) and larger scale CSP tasks (CSP150, CSP200, and CSP300), respectively. The results of the nonlearned heuristic

TABLE III
ATTENTION-DYNAMIC METHOD VERSUS BASELINES ON SMALL-SCALE CSPS. RESULTS OF THE COST AND THE EXECUTION TIME ARE RECORDED IN AVERAGE BY EVALUATING THE MODELS ON 100 RANDOMLY GENERATED INSTANCES. THE GAP IS COMPUTED W.R.T. THE BEST PERFORMING APPROACH

Method	CSP20			CSP50			CSP100		
	Cost	Gap	Time/s	Cost	Gap	Time/s	Cost	Gap	Time/s
LS1	1.87	8.09%	4.24	2.57	0.00%	15.04	3.58	0.00%	88.85
LS2	1.73	0.00%	6.55	2.67	3.89%	21.61	3.69	3.07%	107.98
PN	4.27	146.82%	0.019	7.85	205.45%	0.027	12.05	236.59%	0.042
PN-dynamic	2.15	24.28%	**0.011**	3.29	28.02%	**0.015**	4.68	30.73%	**0.033**
AM	2.23	28.90%	0.014	3.87	50.58%	0.041	5.02	40.22%	0.064
AM-dynamic	**1.89**	**9.25%**	0.014	**2.75**	**7.00%**	0.032	**4.08**	**13.97%**	0.063
AM-dynamic (LS)	**1.85**	**6.94%**	**0.5**	**2.64**	**2.72%**	**0.71**	**3.65**	**1.96%**	**2.66**

TABLE IV
ATTENTION-DYNAMIC METHOD VERSUS BASELINES ON LARGE-SCALE CSPS. RESULTS OF THE COST AND THE EXECUTION TIME ARE RECORDED IN AVERAGE BY EVALUATING THE MODELS ON 100 RANDOMLY GENERATED INSTANCES. THE GAP IS COMPUTED W.R.T. THE BEST PERFORMING APPROACH

Method	CSP150			CSP200			CSP300		
	Cost	Gap	Time/s	Cost	Gap	Time/s	Cost	Gap	Time/s
LS1	4.28	0.00%	331.45	4.86	0.00%	755.77	5.93	0.00%	2637.55
LS2	4.49	4.91%	362.28	5.16	6.17%	695.77	6.34	6.91%	2428.75
PN	16.25	279.67%	0.087	22.54	363.79%	0.117	29.68	400.51%	0.228
PN-dynamic	6.47	51.17%	**0.042**	8.44	73.66%	**0.076**	11.17	88.36%	0.128
AM	6.11	42.76%	0.102	8.28	70.37%	0.162	10.52	77.40%	0.291
AM-dynamic	**5.19**	**21.26%**	0.071	**6.01**	**23.66%**	0.082	**7.62**	**28.50%**	**0.106**
AM-dynamic (LS)	**4.34**	**1.40%**	**7.99**	**4.93**	**1.44%**	**4.93**	**5.98**	**0.84%**	**85.08**

methods are shown in the first part of the tables while the second part shows the results obtained by the pure deep learning models without a posterior local search.

Performances Over Deep Learning Baselines: It is obvious that our method clearly outperforms all deep learning baselines on all CSP tasks. Current state-of-the-art deep learning methods on solving combinatorial problems fail to solve the CSP task, leading to a large optimality gap w.r.t. the classical heuristic solvers. Our method comfortably surpasses these deep learning baselines with a much smaller gap to the heuristic solvers.

Performances Over Heuristic Baselines: We then conduct a simple local search to improve the solutions obtained by our end-to-end model. The results are shown in the third part of Tables III and IV. In addition, it can be seen that conducting a local search can improve the solutions, however, leading to a longer execution time. This observation is consistent with the results reported in [35]. The hybrid method in basis of our attention-dynamic model can achieve significantly closer optimality (no more than 2% on all tasks except CSP20) to the traditional heuristic method such as LS1. Although there is still an optimality gap, our method runs dramatically faster than the heuristic methods. It is about 20 to 40 times faster than the LS1&LS2 with the local search, and thousands times faster than LS1&LS2 without local search. Note that our approach is trained on a different dataset from the test instances. The results are obtained by generalizing the model to unseen CSP tasks with different city locations and dimensions. Thus, it is reasonable that there is a slight performance gap between our method and the heuristic solvers. Also, it is not our goal to defeat the specialized, carefully designed heuristic solvers. Instead, we aim to show the fast solving speed and high generalization ability of the proposed approach.

It should be noticed that the heuristic solvers usually require a stop searching criterion, for example, the cost not changing for a fixed number of epochs. Therefore, with extra searching procedures, their execution time can be longer than expected. The above experiments are conducted with the heuristic methods optimized to their best. However, it would be more interested to see the comparison of running time with all the methods executed to the same level of optimality.

In this context, we, in addition, design controlled experiments to make the comparisons fair and explicit, that is, comparing their execution time by stopping them at the same cost. In specific, the cost obtained by our attention-dynamic (LS) method is set as the benchmark cost and the heuristic methods immediately stop once they reach or surpass the defined cost. Their execution time when stopping at the same cost is recorded. The results are shown in Table V. Clearly, our approach is more than ten times faster than the traditional heuristic solvers if they all reach the same level of optimality. It shows the significantly favorable time complexity of our approach than the traditional solvers.

Performances on Instances From TSPLIB: Furthermore, we evaluate the proposed model's performance on the benchmark test instances that are commonly used in many CSP studies [13]–[15]. The locations of the CSP instances are set as the

TABLE V
COMPARISONS OF THE EXECUTION TIME WHEN ALL THE APPROACHES REACH A SAME COST

Method	CSP50		100-TSP		CSP100		CSP200		CSP300	
	Cost	StopTime/s	Cost	StopTime/s	Cost	StopTime/s	Cost	StopTime/s	Cost	StopTime/s
LS1	2.619	2.35	3.632	27.69	4.332	128.64	4.93	327.59	5.980	1295.65
LS2	2.642	11.14	3.742	71.43	4.476	290.66	5.086	559.48	6.266	2697.78
AM-dynamic (LS)	2.641	**0.78**	3.656	**2.71**	4.337	**8.07**	4.932	**22.14**	5.972	**85.25**

TABLE VI
COMPARISONS OF THE AM-DYNAMIC AND LS1 METHODS ON CSP INSTANCES GENERATED FROM TSPLIB. EXECUTION TIME WHEN ALL THE APPROACHES REACH A SAME COST IS RECORDED

	NC=7			NC=11				NC=7			NC=11		
Methods		LS1	AM-dyna		LS1	AM-dyna	Methods		LS1	AM-dyna		LS1	AM-dyna
Instances	Cost	Time/s	Time/s	Cost	Time/s	Time/s	Instances	Cost	Time/s	Time/s	Cost	Time/s	Time/s
ulysses22	2.253	**0.24**	0.63	1.916	2.71	**0.41**	kroB150	4.318	30.89	**4.53**	3.821	**1.64**	2.55
berlin52	2.812	**0.19**	0.44	2.678	**0.19**	0.39	ch150	4.124	64.6	**5.07**	3.587	8.41	**4.82**
st70	2.925	3.13	**1.05**	3.22	**0.3**	0.45	pr152	4.771	**2.61**	4.1	4.051	26.18	**1.81**
pr76	3.488	5.27	**1.39**	3.158	**0.3**	0.46	u159	4.522	216.34	**7.1**	3.729	99.22	**4.22**
eil76	3.142	30.63	**1.18**	3.051	**0.29**	0.49	rat195	5.146	25.97	**11.22**	4.159	355.49	**8.16**
rat99	3.722	1.16	2.23	3.23	2.09	**1.52**	d198	3.823	91.84	**73.36**	3.474	118.69	**73.97**
rd100	3.622	13.35	**1.61**	2.991	42.01	**1.39**	kroA200	5.09	230.92	**9.12**	4.346	**5.06**	5.33
kroA100	3.738	2.34	**1.63**	2.974	16.25	**1.53**	kroB200	4.845	430.32	**11.03**	3.981	43.11	**6.22**
kroB100	3.704	2.42	**2.38**	2.976	29.29	**1.76**	gr202	4.199	152.49	**39.5**	3.477	94.98	**19.71**
kroC100	3.791	**1.82**	1.54	2.984	47.28	**1.58**	ts225	6.088	533.87	**15.47**	5.061	**4.14**	5.78
kroD100	3.619	5.9	1.41	2.972	13.28	**1.24**	tsp225	5.184	23.79	**15.87**	4.288	19.79	**11.87**
kroE100	3.951	**3.21**	3.36	2.911	2.93	**1.5**	pr226	4.596	19.93	**7.86**	4.102	10.55	**5.02**
eil101	3.726	7.77	**2.8**	2.996	21.05	**1.82**	pr264	4.634	200.33	**158.78**	4.234	237.57	**135.15**
lin105	3.854	23.66	**1.2**	3.495	1.11	1.31	a280	5.763	38.39	**30.6**	4.517	450.32	**20.09**
pr124	3.804	5.89	**2.26**	3.386	23.25	**1.72**	pr299	6.352	29.9	**15.83**	5.076	830.77	**17.23**
ch130	4.551	7.16	**3.52**	3.681	24.69	**2.06**	linhp318	6.143	525.8	**29.78**	5.125	874.1	**23.12**
pr136	4.048	56.11	**3.76**	4.007	**1.03**	1.73	lin318	6.143	514.02	**29.03**	5.125	873.01	**23.35**
pr144	4.902	11	**2.11**	4.519	2.94	**2.08**	rd400	7.08	1730.51	**242.52**	5.613	1182.48	**65.34**
kroA150	4.343	9.08	**4.34**	3.572	39.86	**3.85**	pcb442	7.241	1927.87	**622.64**	5.771	1023.9	**588.03**

TSP locations from the TSPLIB[3] [51]. We still conduct the controlled experiments, that is, comparing the execution time of the compared approaches by stopping them at the same cost. Table VI reports the results. It can be seen that the proposed method surpasses LS1 on most of the TSPLIB instances. On small-scale instances, the heuristic method works well, however, requiring prohibitive execution time on large-scale instances. Note that the model is trained on instances generated from a uniform distribution, whereas the TSPLIB instances are not uniformly distributed. This discrepancy accounts for the slightly worse performance of the model on TSPLIB instances than that on the randomly generated instances.

C. Validation of Generalization

In this part, we analyze the generalization ability of the proposed approach to different types of CSP tasks. The models are trained on CSP instances with $NC = 7$ (each city can cover its seven nearest neighbors), and then are used to solve different types of CSP tasks.

1) We first generalize our model to CSP tasks with different NC. Table VII summarizes the results with $NC = 7$, 11, and 15, respectively.
2) In the above CSP tasks, all cities cover the same number of nearby cities. In addition, we test the CSP task in which each city can cover different number of their nearby cities. To this aim, we generate CSP instances with random NCs ranging from 2 to 15 for each city. The original model trained on instances with $NC = 7$ is used to test this task. Table VIII presents the average results of 100 runs.
3) The same model is then used to solve another category of CSP: each city has a fixed coverage radius within that all other cities are covered. CSP tasks with a fixed coverage radius and variable coverage radius for each city are both tested. The coverage radius is set to a fixed value 0.2 for the first task, and the coverage radius is randomly chosen from a uniform distribution [0, 0.25] for the second task. The results are reported in Table VIII.

The results in Table VII demonstrate that the performance of our approach drops slightly when generalizing to CSP tasks with different NC values. The optimality gap exceeds 3% and 2% for test instances with $NC = 11$ and $NC = 15$, respectively. But our method is still able to produce near-optimal solutions with tiny optimality gap while requiring significantly less execution time compared to the traditional heuristic methods. It is noticed that a large optimality gap is found on CSP100 instances with $NC = 15$. From the optimal cost, we find it is similar with the results on CSP20 instances with $NC = 7$, in which our model also performs the worst (6.94%) when comparing to other test instances. It is, therefore, tentatively concluded that our proposed method performs worse when the length of the optimal tour is short.

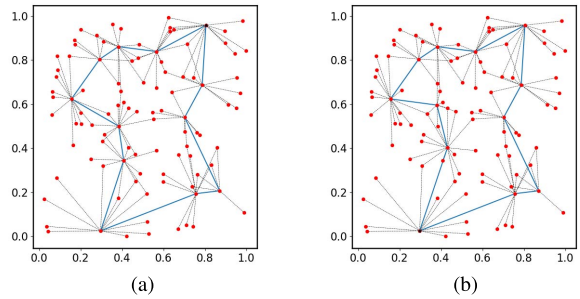
[3]http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/

TABLE VII
ATTENTION-DYNAMIC METHOD VERSUS HEURISTIC BASELINES ON CSP TASKS WITH DIFFERENT NCS. THE AVERAGE RESULTS OF 100 RUNS ARE PRESENTED

NC	Method	CSP100			CSP150			CSP200			CSP300		
		Cost	Gap	Time/s	Cost	Gap	Time/s	Cost	Gap	Time/s	Cost	Gap	Time/s
7	LS1	3.58	0.00%	88.85	4.28	0.00%	331.45	4.86	0.00%	755.77	5.93	0.00%	2637.55
	LS2	3.69	3.07%	107.98	4.49	4.91%	362.28	5.16	6.17%	695.77	6.34	6.91%	2428.75
	AM-dynamic (LS)	3.65	1.96%	2.66	4.34	1.40%	7.99	4.93	1.44%	4.93	5.98	0.84%	85.08
11	LS1	2.94	0.00%	69.6	3.51	0.00%	230.38	4.01	0.00%	557.82	4.93	0.00%	1015.65
	LS2	3.03	3.06%	55.92	3.69	5.13%	144.77	4.22	5.24%	343.93	5.22	5.88%	810.56
	AM-dynamic (LS)	3	2.04%	2.69	3.61	2.85%	5.92	4.15	3.49%	8.29	5.02	1.83%	28.45
15	LS1	2.58	0.00%	57.88	3.09	0.00%	170.93	3.53	0.00%	214.22	4.25	0.00%	1379.24
	LS2	2.64	2.33%	35.69	3.21	3.88%	96.37	3.65	3.40%	425.66	4.44	4.47%	720.74
	AM-dynamic (LS)	3.02	17.05%	1.12	3.17	2.59%	4.22	3.6	1.98%	12.32	4.33	1.88%	35.26

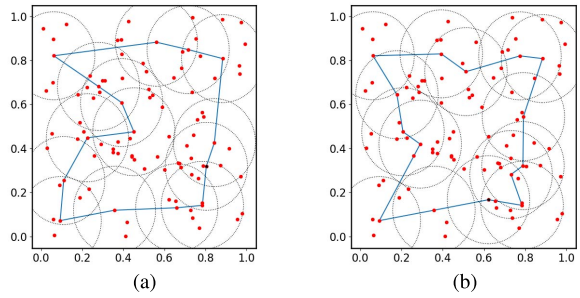
TABLE VIII
ATTENTION-DYNAMIC METHOD VERSUS HEURISTIC BASELINES ON VARIOUS CSP TASKS. THE AVERAGE RESULTS OF 100 RUNS ARE PRESENTED

CSP tasks	Method	CSP50			CSP100			CSP150			CSP200		
		Cost	Gap	Time/s	Cost	Gap	Time/s	Cost	Gap	Time/s	Cost	Gap	Time/s
Varible NCs	LS1	2.34	0.00%	5.62	2.96	0.00%	18.76	3.65	0.00%	93.25	4.14	0.00%	175.68
	LS2	2.36	0.85%	5.33	3.02	2.03%	22.53	3.77	3.29%	67.65	4.27	3.14%	128.65
	AM-dynamic (LS)	2.54	8.55%	0.36	3.01	1.69%	1.56	3.7	1.37%	2.82	4.2	1.45%	6.52
Fixed radius	LS1	2.94	0.00%	17.63	2.96	0.00%	49.1	2.93	0.00%	119.34	2.93	0.00%	237.54
	LS2	3.13	6.46%	21.4	3.04	2.70%	57.58	3.01	2.73%	87.78	2.96	1.02%	124.57
	AM-dynamic (LS)	2.96	0.68%	1.52	2.99	1.01%	1.48	2.99	2.05%	2.42	2.97	1.37%	4.55
Varible radius	LS1	3.71	7.85%	17.84	3.41	0.59%	53.09	3.29	0.00%	81.28	3.17	0.63%	115.38
	LS2	3.76	9.30%	20.53	3.62	6.78%	47.24	3.56	8.21%	63.33	3.46	9.84%	85.21
	AM-dynamic (LS)	3.44	0.00%	1.43	3.39	0.00%	9.56	3.31	0.61%	6.35	3.15	0.00%	3.45



(a) (b)

Fig. 7. Solutions of a CSP100 instance where each city can cover a random number of cities. Tour lengths of the solutions are provided. (a) Attention-dynamic-LS (3.03). (b) LS1 (2.96).



(a) (b)

Fig. 8. Solutions of a CSP100 instance with a fixed cover radius for each city. Tour lengths of the solutions are provided. (a) Attention-dynamic-LS (3.28). (b) LS1 (3.16).

We can see from Table VIII that our method can successfully generalize to various CSP tasks.

For CSP tasks in which different city can cover different number of cities, the model still shows a good performance. The optimality gap of our method can be always within 2% for all CSP instances except CSP50. The worse performance of our approach on CSP50 is consistent with the above experiments. Fig. 7 visualizes the solutions obtained by our method and LS1 on a CSP100 instance with randomly generated NC values. There are only two different cities between the solutions of the two methods, leading to a slight worse performance of our method than LS1.

Notably, our method outperforms LS1 on CSP tasks with variable cover radius. Our method also achieves a closer optimality to LS1 on CSP instances with fixed cover radius. Although the model is not trained on these type of CSP tasks, it can still tackle them successfully. Figs. 8 and 9 visualize the solutions on CSP instances with fixed and variable cover radius, respectively. It can be seen that the obtained solutions of the two compared methods are similar.

It is obvious that our approach can always produce a comparable solution while requiring significantly less execution time. The model only trained on CSP instances of $NC = 7$ can be used to tackle various types of CSP tasks. This generalization is mainly achieved by the dynamic embeddings that we design. Once a city is covered, its probability of being selected is dynamically adjusted by the learned model parameters.

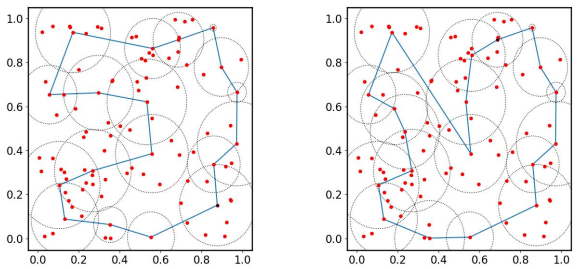


Fig. 9. Solutions of a CSP100 instance where each city has a random covering radius. Tour lengths of the solutions are provided. (a) Attention-dynamic-LS (3.26). (b) LS1 (3.41).

VIII. Conclusion

CSPs arise in numerous application domains such as the disaster planning. However, with the problem dimension increasing in real-world applications, traditional approaches suffer from the limitation of forbidding execution time. Moreover, considerable development efforts are required to design the heuristics. In this context, we introduced a new deep learning approach for the CSP in this study. In this approach, a DNN is used to directly output a permutation of the input. The model is trained using the REINFORCE algorithm with greedy rollout in an unsupervised manner. As an end-to-end approach, this method shows desirable properties of fast solving speed and the ability of generalizing to unseen instances. There is still an optimality gap of our method with respect to traditional solvers. However, in some occasions where the problem is large scale and requires quick decision, the proposed approach is highly desirable as it requires significantly less execution time. Moreover, the esoteric domain knowledge and trial-and-error process needs to design traditional heuristic methods. Years of effort have been devoted to engineer the heuristic rules for CSPs; however, no actual breakthrough is accomplished. Most of the designed heuristics are problem specific and need to be revised once the problem setting changes. Thus, the deep learning approach proposed in this study can be more favorable as it can learn the heuristic from the data by itself and automate the process of decision making.

The proposed method provides a desirable tradeoff of the solution quality and the execution time. It also significantly outperforms the current state-of-the-art deep learning approaches for combinatorial optimization. However, the better solution quality is still the significant advantage of traditional solvers. Improving the solution quality of the deep learning approach is an urgent research perspective in the future. The main challenge is how to handle the dynamic feature of the problem more effectively. Better mechanism should be developed to enable the agent to understand the change of the context when a city is visited and others are covered, and how this dynamic change affects the policy.

References

[1] J. R. Current and D. A. Schilling, "The covering salesman problem," *Transp. Sci.*, vol. 23, no. 3, pp. 208–213, 1989.

[2] S. R. Shariff, N. H. Moin, and M. Omar, "Location allocation modeling for healthcare facility planning in Malaysia," *Comput. Ind. Eng.*, vol. 62, no. 4, pp. 1000–1010, 2012.

[3] D. Reina, S. T. Marin, N. Bessis, F. Barrero, and E. Asimakopoulou, "An evolutionary computation approach for optimizing connectivity in disaster response scenarios," *Appl. Soft Comput.*, vol. 13, no. 2, pp. 833–845, 2013.

[4] N. Vesselinova, R. Steinert, D. F. Perez-Ramirez, and M. Boman, "Learning combinatorial optimization on graphs: A survey with applications to networking," *IEEE Access*, vol. 8, pp. 120388–120416, 2020.

[5] E. L. Lawler and D. E. Wood, "Branch-and-bound methods: A survey," *Oper. Res.*, vol. 14, no. 4, pp. 699–719, 1966.

[6] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2011.

[7] X. Cai *et al.*, "The collaborative local search based on dynamic-constrained decomposition with grids for combinatorial multiobjective optimization," *IEEE Trans. Cybern.*, vol. 51, no. 5, pp. 2639–2650, May 2021.

[8] L. Feng *et al.*, "Explicit evolutionary multitasking for combinatorial optimization: A case study on capacitated vehicle routing problem," *IEEE Trans. Cybern.*, vol. 51, no. 6, pp. 3143–3156, Jun. 2021.

[9] Y.-H. Jia, Y. Mei, and M. Zhang, "A bilevel ant colony optimization algorithm for capacitated electric vehicle routing problem," *IEEE Trans. Cybern.*, early access, Apr. 20, 2021, doi: 10.1109/TCYB.2021.3069942.

[10] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[11] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 2692–2700.

[12] W. Kool, H. van Hoof, and M. Welling, "Attention, learn to solve routing problems!" 2018. [Online]. Available: arXiv:1803.08475.

[13] B. Golden, Z. Naji-Azimi, S. Raghavan, M. Salari, and P. Toth, "The generalized covering salesman problem," *INFORMS J. Comput.*, vol. 24, no. 4, pp. 534–553, 2012.

[14] M. Salari and Z. Naji-Azimi, "An integer programming-based local search for the covering salesman problem," *Comput. Oper. Res.*, vol. 39, no. 11, pp. 2594–2602, 2012.

[15] M. H. Shaelaie, M. Salari, and Z. Naji-Azimi, "The generalized covering traveling salesman problem," *Appl. Soft Comput.*, vol. 24, pp. 867–878, Nov. 2014.

[16] P. Venkatesh, G. Srivastava, and A. Singh, "A multi-start iterated local search algorithm with variable degree of perturbation for the covering salesman problem," in *Harmony Search and Nature Inspired Optimization Algorithms*. Singapore: Springer, 2019, pp. 279–292.

[17] M. Salari, M. Reihaneh, and M. S. Sabbagh, "Combining ant colony optimization algorithm and dynamic programming technique for solving the covering salesman problem," *Comput. Ind. Eng.*, vol. 83, pp. 244–251, May 2015.

[18] S. P. Tripathy, A. Tulshyan, S. Kar, and T. Pal, "A metameric genetic algorithm with new operator for covering salesman problem with full coverage," *Int. J. Control Theory Appl.*, vol. 10, no. 7, pp. 245–252, 2017.

[19] V. Pandiri, A. Singh, and A. Rossi, "Two hybrid metaheuristic approaches for the covering salesman problem," *Neural Comput. Appl.*, vol. 32, pp. 15643–15663, Oct. 2020.

[20] V. Pandiri and A. Singh, "An artificial bee colony algorithm with variable degree of perturbation for the generalized covering traveling salesman problem," *Appl. Soft Comput.*, vol. 78, pp. 481–495, May 2019.

[21] J. J. Hopfield and D. W. Tank, "Neural computation of decisions in optimization problems," *Biol. Cybern.*, vol. 52, no. 3, pp. 141–152, 1985.

[22] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 27, 2014, pp. 3104–3112.

[23] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2009.

[24] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," 2016. [Online]. Available: arXiv:1611.09940.

[25] V. R. Konda and J. N. Tsitsiklis, "Actor–critic algorithms," in *Proc. Adv. Neural Inf. Process. Syst.*, 2000, pp. 1008–1014.

[26] M. Nazari, A. Oroojlooy, L. V. Snyder, and M. Takác, "Deep reinforcement learning for solving the vehicle routing problem," 2018. [Online]. Available: arXiv:1802.04240.

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

[28] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 6348–6358.
[29] V. Mnih *et al.*, "Playing Atari with deep reinforcement learning," 2013. [Online]. Available: arXiv:1312.5602.
[30] A. Mittal, A. Dhawan, S. Manchanda, S. Medya, S. Ranu, and A. Singh, "Learning heuristics over large graphs via deep reinforcement learning," 2019. [Online]. Available: arXiv:1903.03332.
[31] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna, "A note on learning algorithms for quadratic assignment with graph neural networks," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, vol. 1050, 2017, p. 22.
[32] C. K. Joshi, T. Laurent, and X. Bresson, "An efficient graph convolutional network technique for the travelling salesman problem," 2019. [Online]. Available: arXiv:1906.01227.
[33] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 539–548.
[34] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
[35] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau, "Learning heuristics for the tsp by policy gradient," in *Proc. Int. Conf. Integr. Constraint Program. Artif. Intell. Oper. Res.*, 2018, pp. 170–181.
[36] G. A. Croes, "A method for solving traveling-salesman problems," *Oper. Res.*, vol. 6, no. 6, pp. 791–812, 1958.
[37] Q. Ma, S. Ge, D. He, D. Thaker, and I. Drori, "Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning," 2019. [Online]. Available: arXiv:1911.04936.
[38] X. Yu *et al.*, "Set-based discrete particle swarm optimization based on decomposition for permutation-based multiobjective combinatorial optimization problems," *IEEE Trans. Cybern.*, vol. 48, no. 7, pp. 2139–2153, Jul. 2018.
[39] X. Cai, H. Sun, Q. Zhang, and Y. Huang, "A grid weighted sum Pareto local search for combinatorial multi and many-objective optimization," *IEEE Trans. Cybern.*, vol. 49, no. 9, pp. 3586–3598, Sep. 2019.
[40] H. Wang and Y. Jin, "A random forest-assisted evolutionary algorithm for data-driven constrained multiobjective combinatorial optimization of trauma systems," *IEEE Trans. Cybern.*, vol. 50, no. 2, pp. 536–549, Feb. 2020.
[41] K. Li, T. Zhang, and R. Wang, "Deep reinforcement learning for multiobjective optimization," *IEEE Trans. Cybern.*, vol. 51, no. 6, pp. 3103–3114, Jun. 2021.
[42] K. Abe, I. Sato, and M. Sugiyama, "Solving NP-hard problems on graphs by reinforcement learning without domain knowledge," *Simulation*, to be published.
[43] E. Yolcu and B. Póczos, "Learning local search heuristics for Boolean satisfiability," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 7992–8003.
[44] C. K. Joshi, T. Laurent, and X. Bresson, "On learning paradigms for the travelling salesman problem," 2019. [Online]. Available: arXiv:1910.07210.
[45] R. Sato, M. Yamada, and H. Kashima, "Approximation ratios of graph neural networks for combinatorial problems," 2019. [Online]. Available: arXiv:1905.10261.
[46] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev, "Reinforcement learning for combinatorial optimization: A survey," 2020. [Online]. Available: arXiv:2003.03600.
[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[48] K. Cho *et al.*, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," 2014. [Online]. Available: arXiv:1406.1078.
[49] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 229–256, 1998.
[50] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. [Online]. Available: arXiv:1412.6980.
[51] G. Reinelt, "TSPLIB—A traveling salesman problem library," *ORSA J. Comput.*, vol. 3, no. 4, pp. 376–384, 1991.

Kaiwen Li received the B.S. and M.S. degrees from the National University of Defense Technology (NUDT), Changsha, China, in 2016 and 2018, respectively.

He is a Student with the College of System Engineering, NUDT. His research interests include prediction technique, multiobjective optimization, reinforcement learning, data mining, and optimization methods on energy Internet.

Tao Zhang received the B.S., M.S., and Ph.D. degrees from the National University of Defense Technology (NUDT), Changsha, China, in 1998, 2001, and 2004, respectively.

He is a Professor with the College of System Engineering, NUDT. His research interests include multicriteria decision making, optimal scheduling, data mining, and optimization methods on energy Internet.

Rui Wang (Member, IEEE) received the Ph.D. degree in systems engineering from the University of Sheffield, Sheffield, U.K., in 2014, under P. Fleming.

He is currently an Associate Professor with the National University of Defense Technology, Changsha, China. His main research interests include evolutionary computation, multiobjective optimization, and the development of algorithms applicable in practice.

Dr. Wang was a recipient of the Operational Research Society Ph.D. Prize Runners-Up for the Best Ph.D. Dissertation in 2014, and a recipient of the Funds for Distinguished Young Scientists from the Natural Science Foundation of HuNan in 2016.

Yuheng Wang, photograph and biography not available at the time of publication.

Yi Han, photograph and biography not available at the time of publication.

Ling Wang received the B.Sc. degree in automation and the Ph.D. degree in control theory and control engineering from Tsinghua University, Beijing, China, in 1995 and 1999, respectively.

Since 1999, he has been with the Department of Automation, Tsinghua University, where he became a Full Professor in 2008. His current research interests include intelligent optimization and production scheduling.

Prof. Wang was a recipient of the National Natural Science Fund for Distinguished Young Scholars of China, the National Natural Science Award (Second Place) in 2014, the Science and Technology Award of Beijing City in 2008, and the Natural Science Award (First Place in 2003, and Second Place in 2007) nominated by the Ministry of Education of China.