Learning Improvement Heuristics for Solving Routing Problems

Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim

***Abstract*— Recent studies in using deep learning (DL) to solve routing problems focus on construction heuristics, whose solutions are still far from optimality. Improvement heuristics have great potential to narrow this gap by iteratively refining a solution. However, classic improvement heuristics are all guided by handcrafted rules that may limit their performance. In this article, we propose a deep reinforcement learning framework to learn the improvement heuristics for routing problems. We design a self-attention-based deep architecture as the policy network to guide the selection of the next solution. We apply our method to two important routing problems, i.e., the traveling salesman problem (TSP) and the capacitated vehicle routing problem (CVRP). Experiments show that our method outperforms state-of-the-art DL-based approaches. The learned policies are more effective than the traditional handcrafted ones and can be further enhanced by simple diversifying strategies. Moreover, the policies generalize well to different problem sizes, initial solutions, and even real-world data set.**

***Index Terms*— Heuristic algorithms, learning (artificial intelligence), mathematical programming, neural networks, vehicle routing.**

I. Introduction

ROUTING problems, e.g., the traveling salesman problem (TSP) and the capacitated vehicle routing problem (CVRP), are a class of combinatorial optimization problems with numerous real-world applications. Although we solve them regularly in daily life, achieving satisfactory results is still challenging due to their NP-hardness. Classical approaches to routing problems could be categorized into exact methods, approximation methods, and heuristics [1], [2]. Exact methods are often designed based on the branch-and-bound framework, which has the theoretical guarantee of finding the optimal solution but practically limited to small instances for their exponential complexity in the worst case [3], [4]. Approximation methods can find suboptimal solutions with probable worst case guarantees in polynomial time, but they may only exist for specific problems and still be of poor approximation ratios [5], [6]. In practice, heuristics are the most commonly applied approaches for solving routing problems. Despite lacking theoretical guarantee on the solution quality, heuristics often can find desirable solutions within reasonable computational time [7]–[9]. However, the development of heuristics requires substantial trial-and-error, and the performance in terms of solution quality is highly dependent on the intuition and experience of human experts [10].

Recently, there is a growing trend toward applying deep learning (DL) to automatically discover heuristic algorithms for solving routing problems. The underlying rationale comes from two aspects: 1) a class of problem instances may share similar structures and differ only in data that follow a distribution and 2) through supervised or reinforcement learning (RL), DL models can discover the underlying patterns of a given problem class, which could be used to generate alternative algorithms that are better than the human-designed ones [11]. A popular family of methods considers the process of solving routing problems as a sequence generation task and leverages the sequence-to-sequence (Seq2Seq) models [12], [13]. Based on elaborately designed deep structures, such as the recurrent neural network (RNN) [14]–[16] and the attention mechanism [17]–[19], these methods are able to learn heuristics that can produce high-quality solutions.

Though showing promising results, as will be reviewed later, most of the existing DL-based methods focus on learning *construction heuristics*, which creates a complete solution incrementally by adding a node to a partial solution at each step. Despite being comparatively fast, their results still have a relatively large gap to the highly optimized traditional solvers in terms of objective values. To narrow this gap, they often rely on additional procedures (e.g., sampling or beam search) to improve solution quality, which has limited capabilities since they rely on the same trained construction policy.

In this article, rather than learning construction heuristics, we present a framework to directly learn *improvement heuristics*, which improves an initial solution by iteratively performing neighborhood search based on certain local operator, toward the direction of improving solution quality [20]–[22]. Traditional improvement heuristics are guided by handcrafted search policies, which requires substantial domain knowledge

Manuscript received 29 April 2020; revised 2 November 2020; accepted 22 March 2021. Date of publication 1 April 2021; date of current version 2 September 2022. This work was supported by the Singapore Government through the Industry Alignment Fund-Industry Collaboration Projects Grant, conducted at Singtel Cognitive and Artificial Intelligence Laboratory for Enterprises (SCALE@NTU), which is a collaboration between Singapore Telecommunications Ltd., (Singtel) and Nanyang Technological University (NTU). *(Corresponding authors: Wen Song; Zhiguang Cao.)*

Yaoxin Wu is with the SCALE@NTU Corporation Lab, Nanyang Technological University, Singapore 639798 (e-mail: wuyaoxin@ntu.edu.sg).

Wen Song is with the Institute of Marine Science and Technology, Shandong University, Qingdao 266237, China (e-mail: wensong@email.sdu.edu.cn).

Zhiguang Cao is with the Singapore Institute of Manufacturing Technology (SIMTech), Agency for Science Technology and Research (A*STAR), Singapore 138634 (e-mail: zhiguangcao@outlook.com).

Jie Zhang is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798 (e-mail: zhangj@ntu.edu.sg).

Andrew Lim is with the Department of Industrial Systems Engineering and Management, National University of Singapore, Singapore 119077 (e-mail: isealim@nus.edu.sg).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNNLS.2021.3068828.

Digital Object Identifier 10.1109/TNNLS.2021.3068828

to design and may bring only limited improvements to the solutions. In contrast, we exploit deep RL to *automatically* discover better improvement policies. Specifically, we first present an RL formulation for the improvement heuristics, where the policy guides the selection of the next solution. Then, we propose a novel architecture based on self-attention to parameterize the policy, by which we can incorporate a large variety of commonly used pairwise local operators, such as 2-opt and node swap. Finally, we apply the RL framework to two representative routing problems, i.e., TSP and CVRP, and design an actor–critic algorithm to train the policy network.

Extensive results show that our method significantly outperforms existing DL-based ones on TSP and CVRP. The learned policies are indeed more effective than traditional handcrafted rules in guiding the improvement process and can be further enhanced by simple ensemble strategies. Moreover, the policies generalize reasonably well to different problem sizes, initial solutions, and even real-world data set. Note that, similar to previous works [17], [23], our aim is not to outperform highly optimized and specialized traditional solvers but to present a *framework* that can automatically learn good search heuristics without human guidance on different problem types, which is of great practical value when facing real-world problems, and little domain knowledge is available.

II. Related Work

The application of deep neural networks to solve routing problems begins from the seminal work of the pointer network [14]. This RNN-based Seq2Seq model solved TSP in a supervised manner. On top of it, Bello and Pham [15] proposed to use RL to train the pointer network, without the need of labeling the training samples by optimal solutions, which are costly to obtain for NP-hard problems. Nevertheless, the RNN-based encoder in the pointer network inevitably embeds the sequential information of the input, in which the output sequence should be insensitive. Therefore, Nazari *et al.* [16] proposed to linearly map the information of each node to a high-dimensional space with shared parameters so that their model could be applied to CVRP and a stochastic variant.

Inspired by the transformer architecture [24], Kool *et al.* [17] replaced the RNN-based sequential structures in Seq2Seq models with the attention modules in both the encoder and decoder and achieved better performances on both TSP and CVRP. Similarly, the permutation invariant pooling in the transformer architecture was adopted in [18] to solve the multiple TSPs. The attention-based mechanism was also applied for embedding in [19], but its performance relies on an additional 2-opt-based local search.

Different from the Seq2Seq paradigm, Khalil *et al.* [10] adopted the deep Q-Network to train a node selection heuristic that works within a greedy algorithm framework for solving TSP. The representations of internal states (partial solutions) are learned by using a graph neural network (GNN) [25]. In [26], GNN was also used to learn normalized embeddings to reconstruct the adjacent matrix of the TSP graph, in a supervised way. Joshi *et al.* [27] unified some existing deep models for TSP and focused on evaluating the ability of generalizing to large instances.

All the above methods learn construction heuristics and are able to outperform traditional nonlearning-based construction heuristics by a large margin. However, the solution quality is still quite far from optimality. Independently of our work, a NeuRewriter model was proposed recently in [23], which also learns a type of improvement heuristic. On CVRP, it outperforms the best method of learning construction heuristics in [17]. However, it needs to train two policies to separately decide the rewritten region and solution selection and relies on complex node features and customized local operations. In contrast, our method involves only one policy network and uses only raw features and typical local operators that are commonly applied to routing problems. Empirically, our method outperforms NeuRewriter both in solution quality and generalization capability.

Except for the deep models, we note that the idea of learning to solve routing problems could date back to some early attempts in machine learning (ML) [28]. For example, the Hopfield network and Elastic Net were applied in [29] and [30], respectively. Different perspectives to solve CVRP are also proposed in the recent work. For example, Arnold and Sörensen [31] and Lucas *et al.* [32] propose to distinguish near-optimal and nonoptimal solutions by extracting useful features, which could be used to reduce search space to potentially good solutions [33], [34]. However, the above methods are generally instance-dependent or rely on carefully handcrafted features and heuristics. Despite highly optimized results on a certain problem, it takes considerable expertise and time to deploy and tune them for new problems. In contrast, deep models can automatically learn heuristics for different problem types with little domain knowledge. Recent applications of ML and DL to combinatorial optimization problems are reviewed in [11].

III. Preliminaries

Formally, an instance of routing problems can be defined on a graph with a set of n nodes $V = \{1, \ldots, n\}$. Each $v \in V$ has features $x(v)$. A solution $s = (s^1, \ldots, s^I)$ is a tour, i.e., a sequence of nodes with length I, with each element s^i ($i \in \{1, \ldots, I\}$) being a node in V. A feasible tour should satisfy problem-specific constraints, which can be defined as follows for TSP and CVRP.

A. TSP

The tour visits each node exactly once; hence, $I = n$.

B. CVRP

Another node v_d called depot is added to V. The original n nodes represent customers, each with demand $\delta(v)$. We define $\delta(v_d) = 0$. The tour consists of multiple successive routes. Each route starts from the depot and visits a subset of customers in sequence. The constraints include: 1) each customer in the tour must be visited exactly once, and hence, $I > n+1$ and 2) the total customer demand on each route cannot surpass the given capacity D.

In this article, we mainly focus on the 2-D Euclidean TSP and CVRP. Let $c(s^i)$ be the coordinate of the ith location in s. Following previous research, we focus on minimizing the Euclidean distance of the tour s, denoted as $f(s)$.

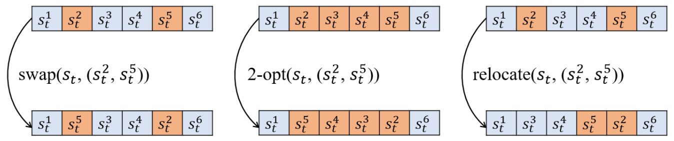

s_t^1 s_t^2 s_t^3 s_t^4 s_t^5 s_t^6
swap$(s_t, (s_t^2, s_t^5))$
s_t^1 s_t^5 s_t^3 s_t^4 s_t^2 s_t^6
s_t^1 s_t^2 s_t^3 s_t^4 s_t^5 s_t^6
2-opt$(s_t, (s_t^2, s_t^5))$
s_t^1 s_t^5 s_t^4 s_t^3 s_t^2 s_t^6
s_t^1 s_t^2 s_t^3 s_t^4 s_t^5 s_t^6
relocate$(s_t, (s_t^2, s_t^5))$
s_t^1 s_t^3 s_t^4 s_t^5 s_t^2 s_t^6

Fig. 1. Three typical pairwise operators for routing problems (left: node swap operator exchanges two locations; middle: 2-opt replaces two links by reversing a segment of locations; and right: relocation puts one location after another location).

C. Improvement Heuristics

Starting from an initial solution s_0, improvement heuristics iteratively replace the current solution s_t at step t with a new solution s_{t+1} picked from neighborhood $\mathcal{N}(s_t)$, toward the direction of minimizing f. The most important parts of improvement heuristics are: 1) the local operator that defines a specific operation on s_t and accordingly yields $\mathcal{N}(s_t)$; 2) the policy used to pick s_{t+1} from $\mathcal{N}(s_t)$; 3) the rules of solution replacement or acceptance; and 4) the termination conditions. Different combinations of the above aspects lead to different schemes of improvement heuristics. For example, hill climbing with the best-improvement strategy picks from $\mathcal{N}(s_t)$ the solution $\bar{s}_{t+1}$ with the smallest f, replaces s_t with $\bar{s}_{t+1}$ only if $f(\bar{s}_{t+1}) < f(s_t)$, and terminates when no such solution exists.

For routing problems, various local operators have been proposed [2]. In this article, we focus on pairwise operators, which transforms a solution s_t to s_{t+1} by performing operation l on a pair of nodes (s_t^i, s_t^j), i.e., $s_{t+1} = l(s_t, (s_t^i, s_t^j))$. The typical pairwise operators are 2-opt that reverses the node sequence between s_t^i and s_t^j, node swap that exchanges s_t^i, s_t^j, and so on. Particularly, Fig. 1 illustrates node swap, 2-opt, and relocation, which are typical pairwise operators for solving routing problems. In general, pairwise operators are fundamental and can be extended to more complex ones. For example, 3-opt and 4-opt can be decomposed into multiple 2-opt operations [35].

Traditional improvement heuristics use handcrafted solution picking policies, which requires substantial domain knowledge to design and could be limited in performance. For example, given the operators, the greedy improvement heuristics (e.g., hill climbing) could quickly get stuck in the local minimum. In this article, we use deep RL to automatically learn high-quality solution picking policies that work in a simple scheme with the following parts: 1) pairwise operators; 2) the "always accept" rule, i.e., the solution picked by the policy will always be accepted, to avoid being stuck in local minima; and 3) a user-specified maximum step T to stop the run. The best solution found in the process is returned after termination. We will show in the experiments that, even with this simple scheme, our method can learn high-quality policies that outperform existing DL-based methods. On the other hand, our method can be extended to guide more complex schemes (e.g., simulated annealing and tabu search). In addition, our method can potentially be applied to other combinatorial problems with sequential solution representations, e.g., scheduling. We plan to tap these potentials in the future.

Remark: In this article, we limit the scope to the basic local search to show that our model can learn more effective policies than conventional handcrafted rules and can be applied to different problems with slight adaptations. We do not investigate other schemes in metaheuristics, e.g., tabu search and guided local search [36]–[39]. However, since most of the metaheuristics are driven by an inner local search process, our model has favorable potential to be equipped with advanced search schemes, which we leave as future work.

IV. Method

We first formulate the process of improvement heuristics as an RL task and then introduce a self-attention-based policy network, followed by the training algorithm. Finally, we provide the details of applying our method to TSP and CVRP.

A. RL Formulation

In this article, we assume that the problem instances are sampled from a distribution $\mathcal{D}$ and use RL to learn the solution picking a policy for the improvement heuristic as we introduced above. To this end, we formulate the underlying Markov decision process (MDP) as follows.

1) State: The state s_t represents a solution to an instance at time step t, i.e., a sequence of nodes. The initial state s_0 is the initial solution to be improved.

2) Action: Since we aim at selecting a solution within the neighborhood structured by pairwise local operators, the action a_t is represented by a node pair (s_t^i, s_t^j), meaning that this node pair is selected from the current state s_t.

3) Transition: The next state s_{t+1} is derived deterministically from s_t by a pairwise local operator, i.e., $s_{t+1} = l(s_t, a_t)$. Taking 2-opt as an example, if $s_t = (\ldots, s_t^i, s_t^{i+1}, \ldots, s_t^{j-1}, s_t^j, \ldots)$ and $a_t = (s_t^i, s_t^j)$, then $s_{t+1} = (\ldots, s_t^j, s_t^{j-1}, \ldots, s_t^{i+1}, s_t^i, \ldots)$.

4) Reward: Our ultimate goal is to improve the initial solution as much as possible within the step limit T. To this end, we design the reward function as follows:

$$r_t = r(s_t, a_t, s_{t+1}) = f\left(s_t^*\right) - \min\left\{f\left(s_t^*\right), f(s_{t+1})\right\} \tag{1}$$

where s_t^* is the best solution found till step t, i.e., the incumbent, which is updated only if s_{t+1} is better, i.e., $f(s_{t+1}) < f(s_t^*)$. Initially, $s_0^* = s_0$. By definition, the reward is positive only when a better solution is found; otherwise, $r_t = 0$. Hence, the cumulative reward (i.e., return) to maximize is expressed as $G_T = \sum_{t=0}^{T-1} \gamma^t r_t$, where γ is the discount factor. When $\gamma = 1$, $G_T = f(s_0) - f(s_T^*)$, which is exactly the improvement over the initial solution s_0.

5) Policy: Starting from s_0, the stochastic policy π picks an action a_t at each step t, which will lead to s_{t+1}, until reaching the step limit T. This process is characterized by a probability chain rule as follows:

$$P(s_T|s_0) = \prod_{t=0}^{T-1} \pi(a_t|s_t). \tag{2}$$

Remark: Note that, in the above MDP, we do not define the terminal states. This is because we intend to apply the trained policy with any user-specified step limit T, in the sense of an anytime algorithm. Hence, we consider the improvement

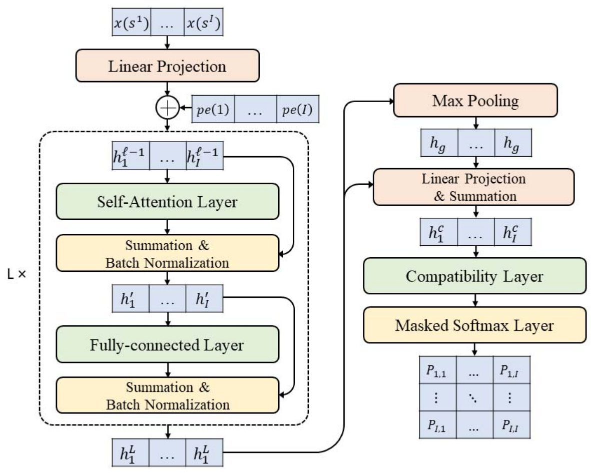


Fig. 2. Architecture of policy network (left: node embedding; right: node pair selection).

process as a continuing task and set $\gamma < 1$. Also, note that the agent is allowed to experience states with poorer quality than the incumbent. Though these "bad" transitions have the lowest immediate reward (0), a higher improvement could be gained in the long term that follows the principle of RL.

B. Policy Network

To learn the stochastic policy π in (2), we parameterize it as a neural network π_θ, where θ refers to the trainable parameters. As visualized in Fig. 2, the network comprises two parts that learn node embedding and node pair selection, respectively. The former part elegantly embeds the nodes in sequence. The latter part adopts the compatibility computation in self-attention to produce a probability matrix of selecting each node pair. Thereby, each element in the matrix refers to the probability of selecting the corresponding node pair for local operation. We would like to note that our node embedding part is similar to the structure of Transformer [24], encoding the sequence by self-attention. Compared to RNNs, self-attention can better capture all node interactions, which is crucial to node pair selection. Also, our architecture can be seen as a graph attention network [40], i.e., a natural structure for coping with node embeddings via self-attention.

1) Node Embedding: Given the current state[1] $s = (s^1, \ldots, s^I)$, the features of each node $x(s^i)$ are first projected to embeddings h_i^0 by a shared linear transformation, with output dimension $d_h = 128$.[2] However, linear mapping alone cannot capture the position of each node in s, which is important since s is a sequence. Hence, we add d_h-dimensional sinusoidal positional encodings $\mathrm{pe}(i, \cdot)$ to node embeddings, refining them as $h_i^0 = h_i^0 + \mathrm{pe}(i, \cdot)$. The sinusoidal positional encodings are a group of vectors defined by sine and cosine functions as follows:

$$\mathrm{pe}(i, d) = \sin\left(i/10\,000^{\frac{\lfloor d/2 \rfloor}{d_h}}\right), \quad \text{if } d \bmod 2 = 0 \tag{3}$$

$$\mathrm{pe}(i, d) = \cos\left(i/10\,000^{\frac{\lfloor d/2 \rfloor}{d_h}}\right), \quad \text{if } d \bmod 2 = 1 \tag{4}$$

where i is the location of node s_i in sequence and d is the dimension. $\lfloor \cdot \rfloor$ and mod mean floor and modulo functions. We also tried relative positional encoding (with embeddings of adjacent nodes added), but it performs worse than the sinusoidal one. To advance the node embeddings $\{h_i^0\}_{i=1}^I$, they are further processed by self-attention layer (**ATT**) and fully connected layer (**FC**), each of which followed by skip connection [41] and batch normalization (**BN**) [42]. This is repeated for L iterations, and each iteration ℓ has its own parameters and does the following computation:

$$h_i' = \mathbf{BN}^\ell\left(h_i^{(\ell-1)} + \mathbf{ATT}_i^\ell\left(h_1^{(\ell-1)}, \ldots, h_I^{(\ell-1)}\right)\right) \tag{5}$$

$$h_i^\ell = \mathbf{BN}^\ell\left(h_i' + \mathbf{FC}^\ell\left(h_i'\right)\right), \quad \ell = 1, \ldots, L \tag{6}$$

where $h_i^{(\ell)}$ is the embedding of node i in iteration ℓ. In this article, we set $L = 3$ for balance between the capacity and memory consumption of the neural network. Since self-attention and fully connected layers are main components of the above structure, we elaborate them in the following.

a) Self-attention layer: Self-attention transforms node embeddings through message passing and aggregation among nodes [25], [40]. We use single-head self-attention here since the multihead version does not lead to significant improvement in our experiments. Given the input matrix $H^{\ell-1} = [h_1^{\ell-1}, \ldots, h_I^{\ell-1}]$ with columns being node embeddings, the self-attention can be expressed as

$$\mathbf{ATT}^\ell\left(H^{\ell-1}\right) = V_\ell \cdot \mathrm{softmax_c}\left(\frac{K_\ell^T Q_\ell}{\sqrt{d_k}}\right) \tag{7}$$

where $Q_\ell = W_\ell^q H^{\ell-1}$, $K_\ell = W_\ell^k H^{\ell-1}$, and $V_\ell = W_\ell^v H^{\ell-1}$ are the *query*, *key*, and *value* matrices of $H^{\ell-1}$, respectively. $W_\ell^q \in \mathbf{R}^{d_q \times d_h}$, $W_\ell^k \in \mathbf{R}^{d_k \times d_h}$, and $W_\ell^v \in \mathbf{R}^{d_v \times d_h}$ are all trainable parameters. Normally, $d_q = d_k$, and d_v determines the output dimension. In our model, $d_q = d_k = d_v = 128$. $\mathrm{softmax_c}(\cdot)$ is a columnwise softmax function so that the output of (7) is the transformed node embeddings.

b) Fully connected layer: This layer transforms each node embedding independently with shared parameters. Here, we only involve one 512-D hidden sublayer with the ReLU activation function. Input and output dimensions retain 128.

2) Node Pair Selection: Given node embeddings $\{h_i^L\}_{i=1}^I$ generated by the previous part, we aggregate them by max-pooling to get the *graph embedding*, i.e., $h_g = \max(\{h_i^L\}_{i=1}^I)$. We then refine each node embedding by converting h_i^L into h_i^c following $h_i^c = W^L h_i^L + W^g h_g$, where W^L, $W^g \in \mathbf{R}^{128\times 128}$. In doing so, the global graph information of an instance is effectively fused into its nodes. Then, we further process the node embeddings through a compatibility layer and a masked softmax layer to get the probability of selecting node pairs.

[1]The step index t is omitted here for better readability.

[2]This dimension is independent of the problem size. We empirically set it as 128, following the previous works of learning models and the original self-attention [15], [17], [24].

a) Compatibility layer: Inspired by Vaswani *et al.* [24], in which the compatibility effectively represents the relations between words in sentences, we adopt it to predict the node pair selection in a solution. While various compatibility functions have been proposed [43], [44], here, we use a multiplicative version given that: 1) its computation is faster since it can be implemented by highly optimized matrix multiplication code and 2) it consumes less GPU memory since it does not need extra neural networks [24]. Given node embeddings $H^c = [h_1^c, \ldots, h_I^c]$, it is computed as the dot product of the query and key matrices, i.e., $Y = K_c^T Q_c$, where K_c and Q_c are computed in a similar way to K_ℓ and Q_ℓ in (7), and the compatibility matrix $Y \in \mathbf{R}^{I\times I}$ reflects the scores of picking each node pair.

b) Masked softmax layer: With certain preprocessing, softmax is applied to the compatibility matrix such that

$$\tilde{Y}_{ij} = \begin{cases} C \cdot \tanh\left(Y_{ij}\right), & \text{if } i \neq j \\ -\infty, & \text{if } i = j \end{cases} \tag{8}$$

$$P = \operatorname{softmax}\left(\tilde{Y}\right) \tag{9}$$

where, in (8), we limit the values in the compatibility matrix within $[-C, C]$ by a tanh function; following Bello and Pham [15], we set $C = 10$ to control the entropy of $\tilde{Y}_{ij}$; and we also mask the diagonal elements since picking pairs of same node is not meaningful. Therefore, the element p_{ij} in P represents the probability of selecting (s^i, s^j) for local operation. Rather than greedily picking the node pair with the maximum probability, we sample the probability matrix P for pair selection in both training and testing.

C. Training Algorithm

We adopt the actor–critic algorithm with the Adam optimizer to train the policy network π_θ. The actor–critic algorithm is a kind of policy gradient method and developed from the REINFORCE [45]. It estimates the cumulative reward by a critic network v_ϕ, which is updated by bootstrapping. In this article, the actor refers to the policy network described above. Our design of v_ϕ is similar to that of actor, except that: 1) mean-pooling is used to obtain the graph embedding and 2) the fused node embeddings are processed by a fully connected layer that is similar to the one used in the policy network but with a single output. We use the **n**-step return for efficient reward propagation and bias–variance tradeoff [46]. In addition, since we do not define the terminal state, it is necessary to bootstrap the value from the time limit state such that the policy for the continuing task can be learned correctly [47]. The complete algorithm is given in Algorithm 1, in which lines from 5 to 20 are used to process a batch of instances in parallel and accumulate their gradients to update the networks.

Algorithm 1 n-Step Actor–Critic (Continuing Task)

```
Input: actor network $\pi_\theta$ with trainable parameters $\theta$;
       critic network $v_\phi$ with trainable parameters $\phi$;
       number of epochs $E$, batches $B$; step limit $T$.
1  for $e = 1, 2, \ldots, E$ do
2      generate $M$ problem instances randomly;
3      for $b = 1, 2, \ldots, B$ do
4          retrieve batch $M_b$; $t \leftarrow 0$;
5          while $t < T$ do
6              reset gradients: $d\theta \leftarrow 0$; $d\phi \leftarrow 0$;
7              $t_s = t$; get state $s_t$;
8              while $t - t_s < \mathbf{n}$ and $t \neq T$ do
9                  sample $a_t$ based on $\pi_\theta(a_t|s_t)$;
10                 receive reward $r_t$ and next state $s_{t+1}$;
11                 $t \leftarrow t + 1$;
12             end
13             $R = v_\phi(s_t)$;
14             for $i \in \{t-1, \ldots, t_s\}$ do
15                 $R \leftarrow r_i + \gamma R$; $\delta \leftarrow R - v_\phi(s_i)$;
16                 $d\theta \leftarrow d\theta + \sum_{M_b} \delta \nabla log\pi_\theta(a_i|s_i)$;
17                 $d\phi \leftarrow d\phi + \sum_{M_b} \delta \nabla v_\phi(s_i)$;
18             end
19             update $\theta$ and $\phi$ by $\frac{d\theta}{|M_b|(t-t_s)}$ and $\frac{d\phi}{|M_b|(t-t_s)}$
20         end
21     end
22 end
```

D. Deployment

In this article, we solve two representative routing problems, i.e., TSP and CVRP. Keeping most of our approach the same, we specialize them for each problem as follows.

1) TSP: Given a solution, the node feature is a 2-D vector that contains the node coordinate, i.e., $x(s^i) = c(s^i)$. Other parts keep the same as introduced above. To avoid solution cycling, the node pair selected at the previous step is masked to forbid the local operation to be reversed. An illustration of our approach solving three five-node TSP instances is displayed in Fig. 3.

2) CVRP: Unlike TSP, solutions to CVRP have varying lengths caused by the times of visiting depot, even with the same number of customers. This makes it hard for batch training and requires additional operations in each step to determine the number and positions of depots in the next solution. To resolve this issue, we add multiple dummy depots to the end of initial solutions such that: 1) we can process a batch of instances using solutions with the same length and 2) the number and positions of depots in a solution (sequence) can be learned automatically, e.g., $s_t = (v_d, s^1, s^2, v_d, \ldots)$ can be turned into $s_{t+1} = (v_d, v_d, s^2, s^1, \ldots)$ by 2-opt, with s_{t+1} being equivalent to $(v_d, s^2, s^1, \ldots)$. To better reflect the local structure, we define the node feature as a 7-D vector $x(s^i) = (c(s^{i-1}), c(s^i), c(s^{i+1}), \delta(s^i))$, i.e., the coordinates of a node and its immediate left and right neighbors in s, along with its demand.[3] Finally, we mask the node pairs in the matrix P that results in infeasible solutions, as well as the one selected at the previous step.

[3]We define the left neighbor of s^1 as s^I and the right neighbor of s^I as s^1, respectively.

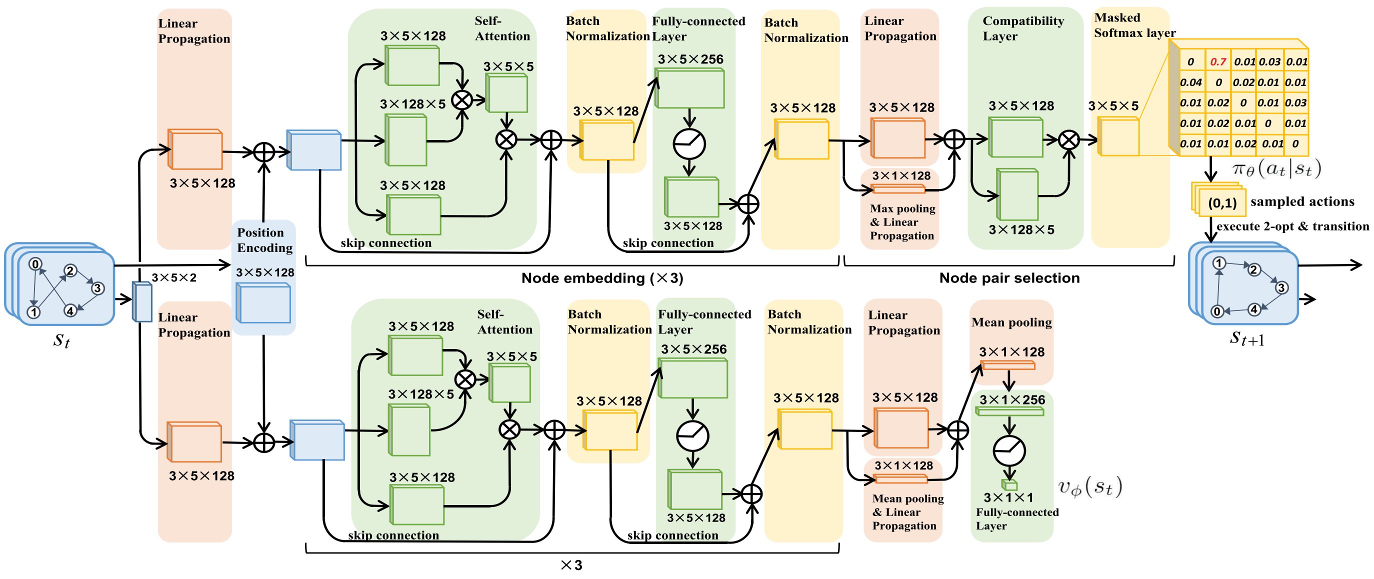


Fig. 3. Illustration of the proposed framework for three five-node TSP instances. Given the current state (tour) s_t, node coordinates and positional encodings as input are processed through the policy network $\pi_\theta(a_t|s_t)$ and the critic network $v_\phi(s_t)$. The action (node pair) a_t is sampled from the output of policy network, and 2-opt is executed to achieve the next state s_{t+1}. Then, during *training*, the reward r_t is computed by (1), and [s_t, a_t, r_t, s_{t+1}, and $v_\phi(s_t)$] is collected to update networks by Algorithm 1; during *inference*, only the trained policy network is used to sample actions and traverse subsequent states.

V. Experimental Results

The instances used in our experiments are Euclidean TSP and CVRP with 20, 50, and 100 nodes, respectively. We call them TSP20, CVRP20, and so on for convenience. We generate the instances following [17], [23], where the coordinates of each node are randomly sampled in the unit square $[0, 1] \times [0, 1]$, with a uniform distribution. For CVRP, the demand of each customer is uniformly sampled from $\{1, \ldots, 9\}$; the capacity D is 30, 40, and 50 for CVRP20, 50, and 100, respectively. More details are given as follows.

TSP In each training epoch for TSP, 10 240 random instances are generated on the fly and split into ten batches. Training starts from random initial solutions, with mean tour distances of 8.48, 19.54, and 37.41 for TSP20, 50, and 100. As mentioned, we model improvement heuristics as continuing tasks. However, we only train the agent for a small step limit $T = 200$ since the rewards in the early stages are denser. We will show later that the trained policies generalize well to unseen initial solutions and much larger T in testing. We set γ to 0.99 and $\mathbf{n}$ for $\mathbf{n}$-step return to 4.

CVRP Due to limited GPU memory, we only generate 3840 instances in each epoch and also split into ten batches. The initial solutions are created using the nearest insertion heuristic adopted in [23], with mean tour distances of 7.74, 13.47, and 20.36 for CVRP20, 50, and 100, which are far from optimality. We add dummy depots to the initial solutions, elongating them to the same length $I^* = 40$, 100, and 125 for CVRP20, 50, and 100.[4] Empirically, for CVRP20, we set $T = 360$ and $\mathbf{n} = 10$; for CVRP50 and CVRP100, we set $T = 480$ and $\mathbf{n} = 12$. For all sizes, we set γ to 0.996.

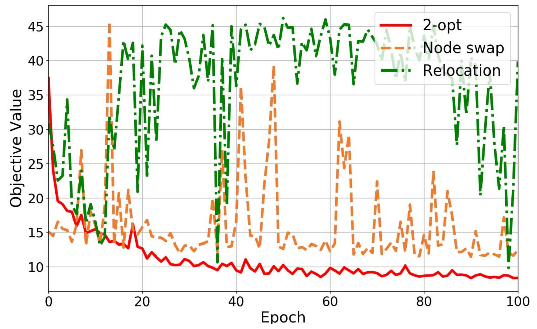
We train 200 epochs for all problems, with an initial learning rate of 10^{-4} and decaying of 0.99 per epoch for convergence. On a single Tesla V100 GPU, each epoch takes on average 8:20 (8 min and 20 s), 16:30, and 31:00 m for TSP20, TSP50, and TSP100 and 20:17, 56:25, and 58:53 m for CVRP20, CVRP50, and CVRP100. We have tried three common pairwise operators, including 2-opt, node swap, and relocation, and the training curves are plotted in Fig. 4, where we can see that 2-opt produces the best validation results. Unless stated otherwise, we only apply 2-opt and use the same settings of initial solutions and additional dummy depots in testing as those in training. Our code in Python and pretrained models will be released soon.

A. Comparison With State-of-the-Art Methods

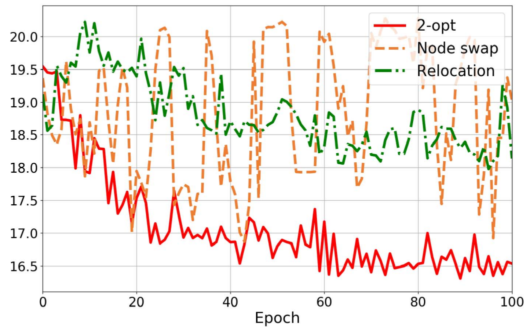
We compare our method with a variety of baselines, including: 1) concorde [48], an efficient exact solver specialized for TSP; 2) LKH3 [21], a well-known heuristic solver that achieves state-of-the-art performance on various routing problems; 3) OR-Tools, a mature and widely used solver for routing problems based on metaheuristics; and 4) state-of-the-art DL-based methods on TSP and CVRP, i.e., the attention model (AM) [17] and NeuRewriter [23], which learns construction and improvement heuristics, respectively.[5] We only evaluate the sampling version of AM, which samples $\mathbb{N}$ solutions using the learned construction policy and is much better than the greedy version. For a fair comparison, we test AM with its default sampling size $\mathbb{N} = 1280$, and also $\mathbb{N} = 5000$, which is the maximum step of our method. For NeuRewriter, since the pretrained model is not provided and training is

[4]The number of depots in a CVRP solution cannot be greater than that of customers n; hence, $I^* = 2n$ is ideal. However, we cannot use $I^* = 200$ for CVRP100 due to GPU memory constraints.

[5]We do not compare with other related methods, such as [10], [15], [16], [19], and [26], since they have already been outperformed by AM on the same benchmark used here [17].



(a)



(b)

Fig. 4. Training curve of 2-opt, node swap, and relocation on (a) TSP100 and (b) CVRP100. All operators are trained for each problem with the same parameters, and the objective values are averaged over a validation set of 200 instances. As shown, the learned policy for 2-opt converges to a significantly lower value than those of node swap and relocation. This might come from the fact that 2-opt takes fewer steps to gain improvements over incumbents (i.e., less sparsity of rewards).

TABLE I
COMPARISON WITH STATE-OF-THE-ART METHODS

	TSP20			TSP50			TSP100			CVRP20			CVRP50			CVRP100		
Methods	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
Concorde	3.83	0.00%	5m	5.69	0.00%	13m	7.76	0.00%	1h	-	-	-	-	-	-	-	-	-
LKH3	3.83	0.00%	42s	5.69	0.00%	6m	7.76	0.00%	25m	6.11	0.00%	1h	10.38	0.00%	5h	15.64	0.00%	9h
OR-Tools	3.86	0.94%	1m	5.85	2.87%	5m	8.06	3.86%	23m	6.46	5.68%	2m	11.27	8.61%	13m	17.12	9.54%	46m
AM ($\mathbb{N}$=1,280)	3.83	0.06%	14m	5.72	0.48%	47m	7.94	2.32%	1.5h	6.26	2.56%	22m	10.61	2.20%	53m	16.17	3.34%	2h
AM ($\mathbb{N}$=5,000)	3.83	0.04%	47m	5.72	0.47%	2h	7.93	2.18%	5.5h	6.25	2.31%	1.5h	10.59	2.01%	3.5h	16.12	3.03%	8h
NeuRewriter	-	-	-	-	-	-	-	-	-	6.15	-	-	10.51	-	-	16.10	-	-
Ours (T=1,000)	**3.83**	**0.03%**	12m	5.74	0.83%	16m	8.01	3.24%	25m	6.16	0.90%	23m	10.71	3.16%	48m	16.30	4.16%	1h
Ours (T=3,000)	**3.83**	**0.00%**	39m	**5.71**	**0.34%**	45m	**7.91**	**1.85%**	1.5h	**6.14**	**0.61%**	1h	10.55	1.65%	2h	16.11	2.99%	3h
Ours (T=5,000)	**3.83**	**0.00%**	1h	**5.70**	**0.20%**	1.5h	**7.87**	**1.42%**	2h	**6.12**	**0.39%**	2h	**10.45**	**0.70%**	4h	**16.03**	**2.47%**	5h

[1] The gap is computed based on the best solutions here given by Concorde and LKH3.
[2] **Bold** results are those outperform the best deep learning based baseline.

prohibitively time-consuming on our machine, we directly report the objective values in the original paper. For each problem size, all methods (except NeuRewriter) are tested on the same 10 000 random instances. For both our method and AM, we divide the instances into ten batches and test each in parallel. We run Concorde, OR-Tools, and LKH3 using the configurations in [17] on a Xeon W-2133 CPU@3.60 GHz. A single thread is used except LKH3, which is relatively slow; hence, we solve 16 instances in parallel. Since run time comparison is hard due to various factors (e.g., Python versus C++ and GPU versus CPU), we follow [17] and report the total time for solving the 10 000 instances.

All results are summarized in Table I, where ours are displayed with step limit $T = 1000$, 3000, and 5000. We can observe that, when $T = 1000$, our method significantly outperforms OR-Tools for both TSP and CVRP with all sizes. As T increases, our results consistently narrow the optimality gaps. AM also benefits from larger $\mathbb{N}$. However, the improvement is not much compared with our method. When $T = 3000$, our method consistently outperforms AM with $\mathbb{N} = 5000$ on all instance sets; for TSP, our method achieves almost the same result as Concorde on TSP20; and for CVRP, our results are on par with NeuRewriter. Though the performance is already good, with additional 2000 steps (i.e., $T = 5000$), our method still can further reduce the optimality gaps and outperforms both baseline deep models with state-of-the-art results on TSP and CVRP. Note that our results still can be improved by increasing T, e.g., results of $T = 8000$ are 7.80 (0.48%) and 15.99 (2.24%) for TSP100 and CVRP100. The above observations show the effectiveness of the continuing design in our RL formulation. That is, despite training with small T, the policies perform fairly well with much larger step limits in testing. In terms of efficiency, the run time of our method is roughly the same order of magnitude as AM. This is well accepted considering the superiority of our method in solution quality. Moreover, with the increase in the problem size, our run time rises much slower than other methods. For example, AM ($\mathbb{N} = 5000$) is faster than ours ($T = 5000$) on TSP20 and CVRP20 but is slower on TSP100 and CVRP100. OR-Tools is also faster than our method ($T = 1000$) on instances with 20 nodes, but, on TSP100 and CVRP100, our method delivers far better solutions than OR-Tools with similar run times.

The above results indicate that learning improvement heuristics instead of construction heuristics could be more promising to solve routing problems. Compared to AM, our method can learn improving solutions more efficiently, e.g., we only use 2×10^6 and 8×10^5 training instances for TSP100 and CVRP100, while AM needs 1×10^8 and 6×10^7. This may

TABLE II
COMPARISON WITH CONVENTIONAL POLICIES

		TSP			CVRP		
	Methods	20	50	100	20	50	100
First	T=1,000	3.84	5.81	8.17	6.18	11.08	17.14
	T=3,000	3.84	5.75	8.04	6.16	10.93	16.93
	T=5,000	3.84	5.73	8.00	6.15	10.87	16.85
Best	T=1,000	3.84	5.75	8.05	6.15	10.79	16.72
	T=3,000	3.84	5.71	7.99	6.14	10.70	16.61
	T=5,000	3.84	5.70	7.94	6.13	10.67	16.55
Ours	T=1,000	**3.83**	5.74	8.01	6.16	10.71	**16.30**
	T=3,000	**3.83**	5.71	**7.91**	6.14	**10.55**	**16.11**
	T=5,000	**3.83**	**5.70**	**7.87**	**6.12**	**10.45**	**16.03**

Bold means our method outperforms the best rule (T=5000).

benefit from the immediate rewards for actions, while, in AM, the reward is delayed, and only received after a complete solution is constructed. Also, it could be difficult for AM to solve large instances since it learns constructing solutions from scratch, while our model is relatively insensitive to the problem size by improving the solution with limited steps.

B. Comparison With Conventional Policies

Compared to the conventional improvement heuristics, the major difference of our method is that the policies of picking the next solution are learned, instead of handcrafted. To show that the automatically learned policies are indeed better than the handcrafted rules, we compare our method with two widely used rules, *first-improvement* and *best-improvement*, which select the first and best cost-reducing solution in the neighborhood, respectively [49]. However, a direct comparison is not fair because, when reaching the local minimum, they cannot pick any solution since no improvement exists. Hence, we augment them with a simple but commonly used strategy, i.e., *restart*, to randomly pick a solution from the whole space when no improvement in the neighborhood can be found. Then, we apply them to the same improvement scheme as our method, i.e., 2-opt with the "always accept" rule and step limit T. We run all policies on the same test sets as those in Section V-A, and the initial solutions are generated in the same way as in training. The results are summarized in Table II. As shown, for the same T, our method consistently outperforms conventional rules in all instance sets, which indicates its potential to achieve better solutions. The advantage of learned policies is more prominent on larger problems, for example, our results with $T = 3000$ already outperform both rules with $T = 5000$ on TSP100, CVRP50, and CVRP100. We can conclude that the learned policies can offer better guidance than conventional ones, especially when facing harder problems. Run time here is not directly comparable since the conventional rules are implemented on the CPU. However, our policy could be more efficient since the neural network directly picks the next solution, without the need of traversing the neighborhood as in the conventional ones.

TABLE III
ENHANCED RESULTS BY DIVERSIFYING

		TSP			CVRP		
	Methods	20	50	100	20	50	100
MP(4)[1]	T=1,000	3.831	5.707	7.897	6.144	10.452	16.110
	T=3,000	3.831	5.701	7.839	6.134	10.426	16.055
	T=5,000	3.831	5.700	7.822	6.123	10.416	16.018
MP(8)	T=1,000	3.831	5.703	7.875	6.134	10.436	16.036
	T=3,000	3.831	5.700	7.824	6.127	10.404	16.000
	T=5,000	3.831	5.699	7.811	6.121	10.395	15.967
MR(4)[2]	T=1,000	3.832	5.707	7.895	6.144	10.482	16.110
	T=3,000	3.831	5.702	7.836	6.132	10.417	15.979
	T=5,000	3.831	5.700	7.821	6.122	10.399	15.923
MR(8)	T=1,000	3.831	5.703	7.866	6.135	10.445	16.057
	T=3,000	3.831	5.700	7.820	6.125	10.393	15.922
	T=5,000	3.831	**5.699**	**7.807**	**6.117**	**10.384**	**15.880**

[1] **MP(#)**: multi-policy strategy with the last # policies.
[2] **MR(#)**: multi-run strategy that runs the final policy # times.

C. Enhancement by Diversifying

The above results are all obtained by running the final learned policy after training only once for each instance. However, this could be less effective in terms of exploring the solution space, e.g., it might suffer from the local minimum for some instances during searching. Here, we show that, by coupling with two simple strategies to diversify the search process, the solution quality of our method could be further improved. The first strategy is multirun, meaning that we directly run the final policy (i.e., the policy obtained after the last training epoch) multiple times. Since we sample the probability matrix P for pair selection during training and set a certain value C in (8) to control the entropy, we avoid the extremely dominant action choice in each step to some extent. Therefore, we could expect to obtain different high-quality solutions by running the same policy multiple times and retrieve the best one as the final solution. The second strategy is multipolicy, for which we run policies of the last several training epochs instead of only the final one on an instance, each generating one solution. Intuitively, multipolicy could provide more diversity than multirun, possibly reaching different regions of the solution space.

Table III shows the performance of the above two strategies. It is clear that the results for all problems are improved with the two strategies, compared to our results in Table II. We can see that, with either strategy, the solution is consistently improved as more runs or policies are used, showing the benefit of diversifying. However, with the same number of runs or policies, multirun outperforms multipolicy for all problems. This is probably because the policies only have small differences since they are in the last phase of training and are not trained to be diverse. Notably, the results of multirun with eight solutions are very close to the optimal solutions for TSP50 and CVRP50, with the gaps of 0.11% and 0.09%. It also narrows the optimality gaps for TSP100 and CVRP100 to 0.56% and 1.52%. We also test multirun on

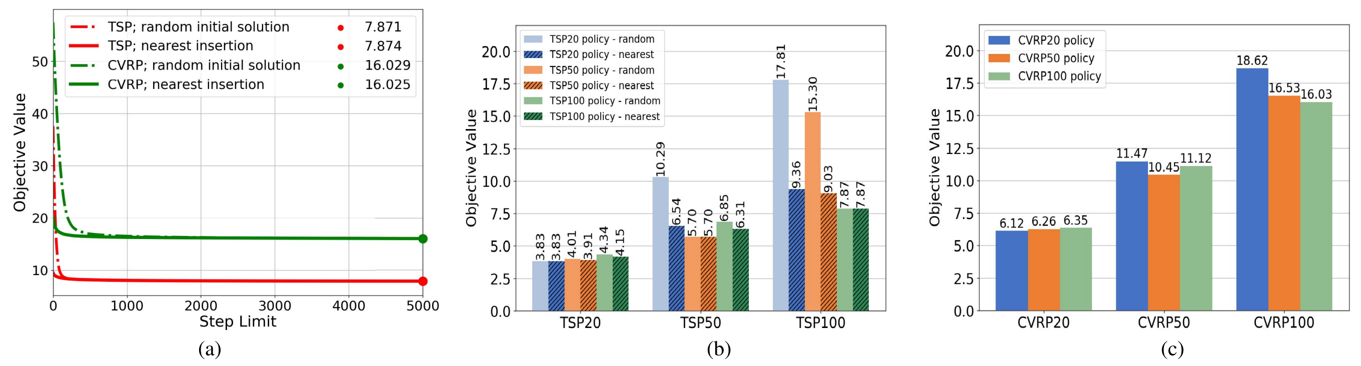


Fig. 5. Generalization analysis. (a) Generalization to initial solutions. (b) Generalization on TSP. (c) Generalization on CVRP.

CVRP100 by generating 16 and 32 solutions, and the objective values further decrease to 15.840 and 15.809 with the optimality gap of 1.24% and 1.08%.

The above analysis shows that both strategies are able to substantially improve the solution quality. Note that these strategies can be effectively parallelized; therefore, little extra time is needed as long as the device has enough memory. Despite the inferior improvement in Table III, we would like to note that cotraining multiple policies to collectively explore solution space is promising for promoting solution quality and efficiency, e.g., training multihead self-attention and keeping each head searching a different solution space. We plan to investigate this in the future.

TABLE IV
GENERALIZATION ON TSP200 AND CVRP200

	TSP200			CVRP200		
Methods	Obj.	Gap	Time	Obj.	Gap	Time
Concorde	10.71	0.00%	21m	-	-	-
LKH3	10.71	0.00%	12m	22.31	0.00%	1.5h
OR-Tools	11.17	4.35%	12m	25.34	12.97%	18m
Ours (T=1,000)	11.51	7.39%	6m	24.72	10.76%	20m
Ours (T=3,000)	11.47	7.01%	18m	24.41	9.39%	1h
Ours (T=5,000)	11.45	6.86%	30m	24.23	8.61%	1.5h

D. Generalization Analysis

Here, we show that the policies learned by our method can generalize to situations unseen in training. All policies here are the ones used in Sections V-A and V-B, without any further tuning. First, we evaluate the sensitivity to different initial solutions on the same problem size. We run the policies trained for TSP100 and CVRP100 on the same test sets used previously, with two types of initial solutions, i.e., generated randomly or by nearest insertion. We plot the average objective values of incumbents against time step T in Fig. 5(a). We can observe that, though the policy for TSP is trained with random initial solutions, it generalizes well to those created by nearest insertion, achieving almost the same quality (7.874 versus 7.871). Similarly, the policy for CVRP, which is trained with initial solutions generated by nearest insertion, can achieve comparable results when beginning from random ones (16.029 versus 16.025). These observations indicate that, for the same problem size, the learned policies generalize well to unseen initial solutions with different qualities.

Furthermore, we evaluate the generalization performance of our method on different problem sizes. Results on TSP are shown in Fig. 5(b). We can see that, when using random initial solutions, though the policies are able to improve (e.g., from 37.41 to 15.30 when the TSP50 policy is used on TSP100), the final results are not very good. This is probably because the quality of random solutions is poor, which makes it hard for such cross-distribution generalization. Hence, we perform the same tests using initial solutions generated by the nearest insertion, and results in Fig. 5(b) show that this leads to relatively good generalization. For the harder problem CVRP, results are shown in Fig. 5(c), based on nearest insertion as in training. We can observe that our policies generalize well to CVRP with different sizes. The policies trained on CVRP50 and CVRP100 outperform OR-Tools on all sizes. In particular, all our results are better than those reported in [23] (e.g., 18.62 versus 18.86 and 16.53 versus 17.33 when CVRP20 and CVRP 50 policy are used on CVRP100, respectively), indicating a stronger generalization ability.

Most of the existing DL-based methods experiment with instances up to 100 nodes [15]–[17], [23]. Here, we further evaluate the generalization performance of our method on larger instances with 200 nodes. Specifically, we generate 1280 random instances for TSP200 and CVRP200, respectively, and set the capacity to 70 for CVRP200. We directly run the models trained for TSP100 and CVRP100 on these larger instances. The results are summarized in Table IV. As shown, the learned policies generalize reasonably well on these instances and even outperform OR-Tools on CVRP200 with only 1000 steps. On the other hand, a recent work in [27] also investigates the zero-shot generalization on TSP200, in which the reported smallest gap is about 8% by using the model trained for TSP100. It is clear that our method can achieve better results, indicating a more desirable generalization capability.

E. Visualization

Here, we give a simple demonstration about what the learned policies based on 2-opt have done along the

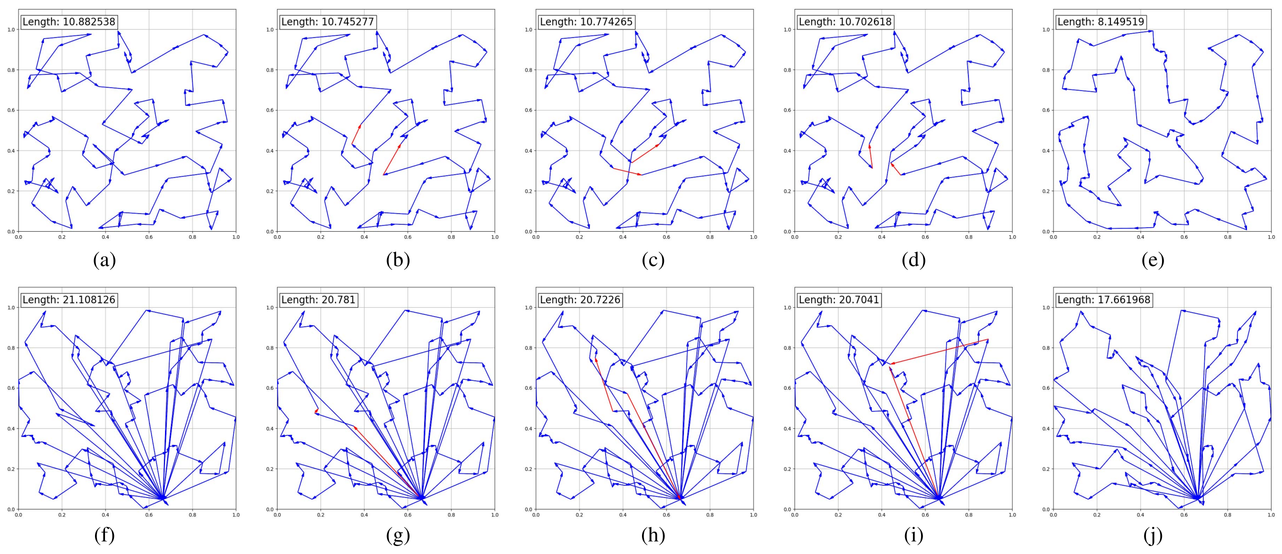


Fig. 6. Visualization of learned policies on TSP and CVRP. (a) Original solution. (b) First 2-opt. (c) Second 2-opt. (d) Third 2-opt. (e) Final solution. (f) Original solution. (g) First 2-opt. (h) Second 2-opt. (i) Third 2-opt. (j) Final solution.

search process. In Fig. 6, we visualize four successive states (solutions) with three actions (2-opt operations) for a TSP100 instance and a CVRP100 instance[6] in Fig. 6(a)–(d) and (f)–(i), respectively. We also provide their final solutions with $T = 200$ in Fig. 6(e) and (j). The red links in each state are two newly added links after two links in the previous state are deleted. From Fig. 6(a)–(d), it can be easily seen that the 2-opt operations effectively decrease the objective value of the TSP100 instance from 10.88 to 10.70, with some cross links deleted. In addition, it verifies that multiple 2-opt could achieve complex operations, as mentioned in Section III. For example, the second and third 2-opts together complete a 3-opt operation. In Fig. 6(e), we can find that the final solution is reasonably good without obvious cross links. For the CVRP100 instance in Fig. 6(f)–(j), we can also observe that the learned policy constantly decreases the objective value, and the final solution has few cross links in the routes. Different from the TSP100 instance, the learned policy on CVRP100 involves the interroute and intraroute operations. For example, the first action is an intraroute operation, deleting and adding links in a single route as in TSP; the second and third actions are interroute operations, generating new routes by destroying some previous routes.

The visualization shows the potential of our method in learning policies that can execute various effective operations based on the 2-opt operator. We would like to note that it is promising to represent multiple operators and, thus, learn policies that simultaneously specify the local operation and next solution. Our policy network in this article can be easily extended to empower multiple operators, for example, using structures similar to the multihead self-attention [24]. A more advanced approach is to use hierarchical RL [50] to decompose the selection of operator and next solution and make decisions for them alternately.

F. Test on Real-World Data Set

We further verify that our learned policies, though trained using synthetic data, performs reasonably well on instances from public benchmarks TSPlib[7] [51] and CVRPlib,[8] [52] which contains real-world problem instances. Note that these instances may follow distributions that are completely different from those we used in training, in aspects such as node location patterns, customer demands, and vehicle capacities.

For TSPlib, we first directly run our policy trained on TSP100 with $T = 3000$, on the 36 symmetric and Euclidean instances up to 300 nodes, and compare with the AM policy also trained on TSP100. As shown in Table V, our TSP100 policy (denoted as Ours) performs much better than those of AM with 1280 and 5000 samples, indicating a stronger generalization ability on these instances. Besides the much smaller average optimality gap, our policy outperforms AM ($\mathbb{N} = 5000$) on 75% (27 out of 36) of these TSPlib instances. On the other hand, all learned policies are generally inferior to OR-Tools and 2-opt[9] when directly used. This is reasonable because, in general, achieving good out-of-distribution generalization is very hard for ML models [27], [53], [54]. This could potentially be alleviated by adapting the trained policy to the (different) testing distribution, via transfer or few-shot learning, but is beyond the scope of this article. However, here, we show that the out-of-distribution performance of our policy could be improved by a simple adjustment. Specifically, instead of picking only one action in each step, we sample $\mathbb{M}$ actions from the policy to obtain multiple solutions and pick the best one. We also allow the restart strategy. For TSPlib instances, we set $\mathbb{M} = 1000$ and keep $T = 3000$. As shown in Table V, our policy with these slight modifications (denoted as Ours-M) significantly reduces

[6]Please refer to https://youtu.be/97ZXp9zSEK8 for a demonstration of solving the TSP100 and CVRP100 instances.

[7]http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html

[8]http://vrp.galgos.inf.puc-rio.br/index.php/en/

[9]Here, we only report the result of the best-improvement 2-opt with *restart*, which performs better than the first-improvement version.

TABLE V

GENERALIZATION ON TSPLIB

Instance	Opt.	OR-Tools	2-OPT	AM ($\mathbb{N}$=1,280)	AM ($\mathbb{N}$=5,000)	Ours (T=3,000)	Ours-M ($\mathbb{M}$=1,000)
eil51	426	436	440	436	435	438	**431***
berlin52	7,542	7,945	7,834	7,717	**7,668***	8,020	7,736
st70	675	683	707	691	690	706	**681***
eil76	538	561*	570	564	563	575	**563**
pr76	108,159	111,104	111,486	111,605	111,250	109,668	**109,638***
rat99	1,211	1,232*	1,319	1,483	1,394	1,419	**1,314**
KroA100	21,282	21,448*	21,940	44,385	38,200	25,196	**21,725**
KroB100	22,141	23,006*	23,537	35,921	35,511	26,563	**23,421**
KroC100	20,749	21,583	22,194	31,290	30,642	25,343	**21,407***
KroD100	21,294	21,636*	23,264	34,775	32,211	24,771	**22,359**
KroE100	22,068	22,598*	23,786	28,596	27,164	26,903	**22,794**
rd100	7,910	8,189	8,731	8,169	8,152	7,915	**7,915***
eil101	629	664	679	668	667	658	**658***
lin105	14,379	14,824	15,641	53,308	51,325	18,194	**14,736***
pr107	44,303	45,072*	46,667	208,531	205,519	53,056	**46,018**
pr124	59,030	62,519	61,976	183,858	167,494	66,010	**60,786***
bier127	118,282	122,733	122,216*	210,394	207,600	142,707	**122,395**
ch130	6,110	6,284*	6,643	6,329	**6,316**	7,120	6,409
pr136	96,772	102,213*	103,455	103,470	**102,877**	105,618	102,900
pr144	58,537	59,286*	60,671	225,172	183,583	71,006	**59,354**
ch150	6,528	6,729*	6,800	6,902	6,877	7,916	**6,760**
KroA150	26,524	27,592*	28,672	44,854	42,335	31,244	**27,720**
KroB150	26,130	27,572	27,904	45,374	43,114	31,407	**27,542***
pr152	73,682	75,834	77,159	106,180	103,110	85,616	**75,282***
u159	42,080	45,778	46,010	124,951	115,372	51,327	**45,308***
rat195	2,323	2,389*	2,607	3,798	3,661	2,913	**2,553**
d198	15,780	15,963*	16,544	78,217	68,104	17,962	**16,568**
KroA200	29,368	29,741*	32,220	62,013	58,643	35,958	**31,426**
KroB200	29,437	30,516*	32,077	54,570	50,867	36,412	**31,511**
ts225	126,643	128,564*	133,326	141,951	141,628	158,748	**130,584**
tsp225	3,916	4,046*	4,188	25,887	24,816	4,701	**4,196**
pr226	80,369	82,968*	83,160	105,724	101,992	97,348	**83,756**
gil262	2,378	2,519*	2,592	2,695	2,693	2,963	**2,555**
pr264	49,135	51,954*	54,019	361,160	338,506	65,946	**52,133**
a280	2,579	2,713*	29,02	13,087	11,810	2,989	**2,858**
pr299	48,191	49,447*	53,855	513,809	513,673	59,786	**5,2811**
Avg. Gap	0	3.46%	6.97%	146.12%	133.54%	17.12%	4.70%

[1] **Bold** means the best among four learning based methods.
[2] * means the best except the optimal solution.

TABLE VI

GENERALIZATION ON CVRPLIB

Instance	Opt.	OR-Tools	2-OPT	AM ($\mathbb{N}$=1280)	AM ($\mathbb{N}$=5000)	Ours (T=5,000)	Ours-M ($\mathbb{M}$=100)
X-n101-k25	27,591	29,405	29,708	39,437	37,702	29,716	**29,136***
X-n106-k14	26,362	27,343	27,129	28,320	28,473	27,642	**27,108***
X-n110-k13	14,971	16,149	16,028	15,627	**15,443***	15,927	15,629
X-n115-k10	12,747	13,320*	14,434	13,917	13,745	14,445	**13,409**
X-n120-k6	13,332	14,242	14,548	14,056	**13,937***	15,486	14,073
X-n125-k30	55,539	58,665	67,433	75,681	75,067	60,423	**58,157***
X-n129-k18	28,940	31,361	30,261	30,399	**30,176***	32,126	30,279
X-n134-k13	10,916	13,275	12,237	13,795	13,619	12,669	**11,885***
X-n139-k10	13,590	15,223	14,724	14,293	**14,215***	15,627	14,256
X-n143-k7	15,700	17,470	16,716*	17,414	17,397	18,872	**16,738**
X-n148-k46	43,448	46,836	44,901*	79,611	79,514	50,563	**45,012**
X-n153-k22	21,220	22,919	21,885*	38,423	37,938	26,088	**22,182**
X-n157-k13	16,876	17,309*	17,683	21,702	21,330	19,771	**17,484**
X-n162-k11	14,138	15,030	15,192	15,108	15,085	16,847	**14,882***
X-n167-k10	20,557	22,477	22,629	22,365	22,285	24,365	**22,258***
X-n172-k51	45,607	50,505	47,640	86,186	87,809	51,108	**47,595***
X-n176-k26	47,812	52,111	50,440*	58,107	58,178	57,131	**50,759**
X-n181-k23	25,569	26,321	26,107	27,828	27,520	27,173	**26,102***
X-n186-k15	24,145	26,017	26,301	25,917	**25,757***	28,422	25,991
X-n190-k8	16,980	18,088*	18,202	37,820	36,383	20,145	**18,132**
X-n195-k51	44,225	50,311	46,266	79,594	79,276	51,763	**46,204***
X-n200-k36	58,578	61,009*	61,848	78,679	76,477	64,200	**61,072**
Avg. Gap	0	8.06%	7.28%	32.97%	31.62%	14.27%	**5.19%**

[1] **Bold** means the best among four learning based methods.
[2] * means the best except the optimal solution.

the average optimality gap to 4.70%, which is smaller than that of 2-opt and close to OR-Tools. Also, our method finds the best solutions on 11 instances, such as pr76 (1.39%), rd100 (0.06%), and lin105 (2.48%). Moreover, the average gaps of instances with 0–100, 101–200, and 201–300 nodes are 3.21%, 4.84%, and 6.92%, respectively, showing that the quality of our solution does not degrade fast with the increase of the problem size.

For CVRPlib, we first directly test the policy trained on CVRP100 with $T = 5000$ on 22 instances with sizes between 101 and 200, each of which is generated following different depot positioning (Central, Eccentric, and Random), customer positioning (Random and Clustered), and demand distribution (small or large variance). The results in Table VI show that our policy significantly outperforms the AM method. Our policy achieves an average optimality gap that is more than two times smaller than AM ($\mathbb{N} = 5000$) and performs better on a majority of (13 out of 22) these instances. In addition, when we allow sampling 100 actions ($\mathbb{M} = 100$) and keep $T = 5000$, our average optimality gap decreases to 5.19% that is far better than those of 2-opt and OR-Tools. Our policy also achieves the best solutions on 9 instances, more than other methods. Moreover, the quality degradation of our solution is slow since the average gaps on the instances with 101–150 and 151–200 nodes are 5.17% and 5.22%, respectively.

VI. CONCLUSION AND FUTURE WORK

This article proposes a deep RL framework to automatically learn improvement heuristics for routing problems. We design a novel neural architecture based on self-attention to enable learning with pairwise local operators. Empirically, our method outperforms state-of-the-art deep models on both TSP and CVRP and further narrows the gap to highly optimized solvers. The learned policies generalize well to different initial solutions and problem sizes and give reasonably good solutions on real-world data sets. In the future, we will further analyze the influence of each component in our model on the performance, through ablation studies.

We would like to note that our method has great potentials in learning a variety of more complex improvement heuristics. First, we could extend the current framework to support multiple operators in the sense that the policy learns to pick both the operator and the next solution. This could be achieved by using techniques, such as multihead self-attention or hierarchical RL. Second, despite the simple search scheme used in this article, our framework can be applied to learn better solution picking policies for more advanced search schemes, such as simulated annealing, tabu search, large neighborhood search, and LKH. Finally, our method is applicable to other important types of combinatorial optimization problems, e.g., scheduling. We plan to investigate these possibilities in the future.

REFERENCES

[1] G. Gutin and A. P. Punnen, *The Traveling Salesman Problem and its Variations*, vol. 12. Boston, MA, USA: Springer, 2006.
[2] P. Toth and D. Vigo, *Vehicle Routing: Problems, Methods, and Applications*. Philadelphia, PA, USA: SIAM, 2014.
[3] G. Laporte and Y. Nobert, "A branch and bound algorithm for the capacitated vehicle routing problem," *Oper.-Res.-Spektrum*, vol. 5, no. 2, pp. 77–85, Jun. 1983.
[4] J. Lysgaard, A. N. Letchford, and R. W. Eglese, "A new branch-and-cut algorithm for the capacitated vehicle routing problem," *Math. Program.*, vol. 100, no. 2, pp. 423–445, Jun. 2004.

[5] N. Bansal, A. Blum, S. Chawla, and A. Meyerson, "Approximation algorithms for deadline-TSP and vehicle routing with time-windows," in *Proc. 36th Annu. ACM Symp. Theory Comput. (STOC)*, 2004, pp. 166–174.

[6] A. Das and C. Mathieu, "A quasi-polynomial time approximation scheme for Euclidean capacitated vehicle routing," in *Proc. 21st Annu. ACM-SIAM Symp. Discrete Algorithms*, Jan. 2010, pp. 390–403.

[7] R. Hassin and A. Keinan, "Greedy heuristics with regret, with application to the cheapest insertion algorithm for the TSP," *Oper. Res. Lett.*, vol. 36, no. 2, pp. 243–246, Mar. 2008.

[8] T. Pichpibul and R. Kawtummachai, "An improved Clarke and Wright savings algorithm for the capacitated vehicle routing problem," *ScienceAsia*, vol. 38, no. 3, pp. 307–318, Jun. 2012.

[9] S. Ropke and D. Pisinger, "An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows," *Transp. Sci.*, vol. 40, no. 4, pp. 455–472, Nov. 2006.

[10] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Proc. 31st Conf. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 6348–6358.

[11] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: A methodological tour d'horizon," 2018, *arXiv:1811.06128*. [Online]. Available: http://arxiv.org/abs/1811.06128

[12] Y. Keneshloo, T. Shi, N. Ramakrishnan, and C. K. Reddy, "Deep reinforcement learning for sequence-to-sequence models," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 31, no. 7, pp. 2469–2489, Jul. 2020.

[13] B. Zhang, D. Xiong, J. Xie, and J. Su, "Neural machine translation with GRU-gated attention model," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 31, no. 11, pp. 4688–4698, Nov. 2020.

[14] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Proc. 29th Conf. Neural Inf. Process. Syst. (NIPS)*, 2015, pp. 2692–2700.

[15] I. Bello and H. Pham, "Neural combinatorial optimization with reinforcement learning," in *Proc. 5th Int. Conf. Learn. Represent. (ICLR)*, 2017.

[16] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác, "Reinforcement learning for solving the vehicle routing problem," in *Proc. 32nd Conf. Neural Inf. Process. Syst. (NIPS)*, 2018, pp. 9839–9849.

[17] W. Kool, H. van Hoof, and M. Welling, "Attention, learn to solve routing problems!" in *Proc. 7th Int. Conf. Learn. Represent. (ICLR)*, 2019.

[18] Y. Kaempfer and L. Wolf, "Learning the multiple traveling salesmen problem with permutation invariant pooling networks," 2018, *arXiv:1803.09621*. [Online]. Available: http://arxiv.org/abs/1803.09621

[19] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau, "Learning heuristics for the TSP by policy gradient," in *Proc. 15th Int. Conf. Integr. Constraint Program., Artif. Intell., Oper. Res. (CPAIOR)*, 2018, pp. 170–181.

[20] D. S. W. Lai, O. C. Demirag, and J. M. Y. Leung, "A tabu search heuristic for the heterogeneous vehicle routing problem on a multigraph," *Transp. Res. E, Logistics Transp. Rev.*, vol. 86, pp. 32–52, Feb. 2016.

[21] K. Helsgaun, "An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems," Roskilde Univ., Roskilde, Denmark, Tech. Rep., 2017. [Online]. Available: https://forskning.ruc.dk/en/publications/an-extension-of-the-lin-kernighan-helsgaun-tsp-solver-for-constra

[22] L. Wei, Z. Zhang, D. Zhang, and S. C. H. Leung, "A simulated annealing algorithm for the capacitated vehicle routing problem with two-dimensional loading constraints," *Eur. J. Oper. Res.*, vol. 265, no. 3, pp. 843–859, Mar. 2018.

[23] X. Chen and Y. Tian, "Learning to perform local rewriting for combinatorial optimization," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 6278–6289.

[24] A. Vaswani *et al.*, "Attention is all you need," in *Proc. 31st Conf. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 5998–6008.

[25] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021.

[26] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna, "A note on learning algorithms for quadratic assignment with graph neural networks," *stat*, vol. 1050, p. 22, Jun. 2017.

[27] C. K. Joshi, Q. Cappart, L.-M. Rousseau, T. Laurent, and X. Bresson, "Learning TSP requires rethinking generalization," 2020, *arXiv:2006.07054*. [Online]. Available: http://arxiv.org/abs/2006.07054

[28] K. A. Smith, "Neural networks for combinatorial optimization: A review of more than a decade of research," *INFORMS J. Comput.*, vol. 11, no. 1, pp. 15–34, Feb. 1999.

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

[30] R. Durbin and D. Willshaw, "An analogue approach to the travelling salesman problem using an elastic net method," *Nature*, vol. 326, no. 6114, p. 689, 1987.

[31] F. Arnold and K. Sörensen, "What makes a VRP solution good? The generation of problem-specific knowledge for heuristics," *Comput. Oper. Res.*, vol. 106, pp. 280–288, Jun. 2019.

[32] F. Lucas, R. Billot, and M. Sevaux, "A comment on 'What makes a VRP solution good? The generation of problem-specific knowledge for heuristics,"' *Comput. Oper. Res.*, vol. 110, pp. 130–134, Oct. 2019.

[33] F. Arnold, Ì. Santana, K. Sörensen, and T. Vidal, "PILS: Exploring high-order neighborhoods by pattern mining and injection," 2019, *arXiv:1912.11462*. [Online]. Available: http://arxiv.org/abs/1912.11462

[34] F. Lucas, R. Billot, M. Sevaux, and K. Sörensen, "Reducing space search in combinatorial optimization using machine learning tools," in *Proc. Int. Conf. Learn. Intell. Optim.* Cham, Switzerland: Springer, 2020, pp. 143–150.

[35] K. Helsgaun, "General K-opt submoves for the Lin–Kernighan TSP heuristic," *Math. Program. Comput.*, vol. 1, nos. 2–3, pp. 119–163, 2009.

[36] C. D. Tarantilis, "Solving the vehicle routing problem with adaptive memory programming methodology," *Comput. Oper. Res.*, vol. 32, no. 9, pp. 2309–2327, Sep. 2005.

[37] C. D. Tarantilis and C. T. Kiranoudis, "BoneRoute: An adaptive memory-based method for effective fleet management," *Ann. Oper. Res.*, vol. 115, nos. 1–4, pp. 227–241, 2002.

[38] C. Voudouris and E. Tsang, "Guided local search and its application to the traveling salesman problem," *Eur. J. Oper. Res.*, vol. 113, no. 2, pp. 469–499, Mar. 1999.

[39] F. Glover, "Tabu search and adaptive memory programming—Advances, applications and challenges," in *Interfaces in Computer Science and Operations Research*. Boston, MA, USA: Springer, 1997, pp. 1–75.

[40] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. 6th Int. Conf. Learn. Represent. (ICLR)*, 2018.

[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[42] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. 32nd Int. Conf. Mach. Learn. (ICML)*, 2015, pp. 448–456.

[43] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, 2015.

[44] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2015, pp. 1412–1421.

[45] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 229–256, May 1992.

[46] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," in *Proc. 33rd Int. Conf. Mach. Learn. (ICML)*, 2016, pp. 1928–1937.

[47] F. Pardo, A. Tavakoli, V. Levdik, and P. Kormushev, "Time limits in reinforcement learning," in *Proc. 35th Int. Conf. Mach. Learn. (ICML)*, 2018, pp. 4042–4051.

[48] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. (2006). *Concorde TSP Solver*. [Online]. Available: http://www.math.uwaterloo.ca/tsp/concorde

[49] P. Hansen and N. Mladenović, "First vs. best improvement: An empirical study," *Discrete Appl. Math.*, vol. 154, no. 5, pp. 802–817, Apr. 2006.

[50] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 3675–3683.

[51] G. Reinelt, "TSPLIB—A traveling salesman problem library," *ORSA J. Comput.*, vol. 3, no. 4, pp. 376–384, 1991.

[52] E. Uchoa, D. Pecin, A. Pessoa, M. Poggi, T. Vidal, and A. Subramanian, "New benchmark instances for the capacitated vehicle routing problem," *Eur. J. Oper. Res.*, vol. 257, no. 3, pp. 845–858, Mar. 2017.

[53] Y. Sun, X. Wang, Z. Liu, J. Miller, A. A. Efros, and M. Hardt, "Test-time training with self-supervision for generalization under distribution shifts," 2019, *arXiv:1909.13231*. [Online]. Available: http://arxiv.org/abs/1909.13231

[54] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, "Quantifying generalization in reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 1282–1289.

Yaoxin Wu received the B.Eng. degree in traffic engineering from Wuyi University, Jiangmen, China, in 2015, and the M.Eng. degree in control engineering from the Guangdong University of Technology, Guangzhou, China, in 2018. He is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, Nanyang Technological University, Singapore.

He is currently a Research Associate with the Singtel Cognitive and Artificial Intelligence Lab for Enterprises (SCALE@NTU), Nanyang Technological University. His research interests include vehicle routing, graph neural networks, deep learning, and deep reinforcement learning.

Wen Song received the B.S. degree in automation and the M.S. degree in control science and engineering from Shandong University, Jinan, China, in 2011 and 2014, respectively, and the Ph.D. degree in computer science from Nanyang Technological University, Singapore, in 2018.

He was a Research Fellow with the Singtel Cognitive and Artificial Intelligence Lab for Enterprises (SCALE@NTU), Nanyang Technological University. He is currently an Associate Research Fellow with the Institute of Marine Science and Technology, Shandong University. His current research interests include artificial intelligence, planning and scheduling, multiagent systems, and operations research.

Zhiguang Cao received the B.Eng. degree in automation from the Guangdong University of Technology, Guangzhou, China, the M.Sc. degree in signal processing from Nanyang Technological University (NTU), Singapore, and the Ph.D. degree from the Interdisciplinary Graduate School, Nanyang Technological University.

He was a Research Assistant Professor with the Department of Industrial Systems Engineering and Management, National University of Singapore, Singapore, and a Research Fellow with the Future Mobility Research Lab, NTU. He is currently the Scientist with the Singapore Institute of Manufacturing Technology (SIMTech), Agency for Science Technology and Research (A*STAR), Singapore. His research interest includes neural combinatorial optimization.

Jie Zhang received the Ph.D. degree from the Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada, in 2009.

He is currently an Associate Professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. He is also an Associate Professor with the Singapore Institute of Manufacturing Technology, Singapore.

Dr. Zhang was a recipient of the Alumni Gold Medal at the 2009 Convocation Ceremony; the Gold Medal is awarded once a year to honor the top Ph.D. graduate from the University of Waterloo. During his Ph.D. study, he held the prestigious NSERC Alexander Graham Bell Canada Graduate Scholarship rewarded for top Ph.D. students across Canada. His papers have been published by top journals and conferences and received several best paper awards. He is also active in serving research communities.

Andrew Lim received the Ph.D. degree in computer science from the University of Minnesota, Minneapolis, MN, USA, in 1992.

He is currently a Professor with the Department of Industrial Systems Engineering and Management, National University of Singapore (NUS), Singapore. Before he was recruited by NUS under The National Research Foundation's Returning Singaporean Scientists Scheme in 2016, he spent more than a decade in Hong Kong where he held professorships in The Hong Kong University of Science and Technology and the City University of Hong Kong. His works have been published in key journals, such as *Operations Research and Management Science*, and disseminated via international conferences and professional seminars.