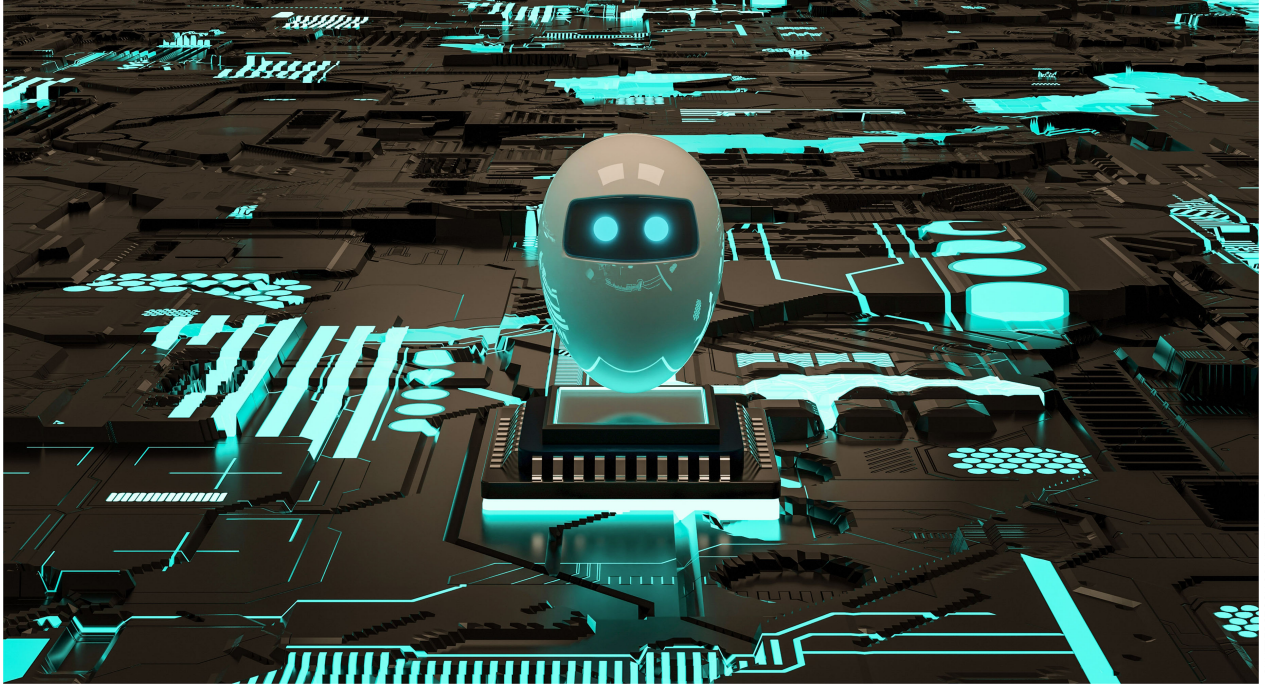

©SHUTTERSTOCK.COM/METHU DAS

How Good is Neural Combinatorial Optimization? A Systematic Evaluation on the Traveling Salesman Problem

Shengcai Liu, Yu Zhang, Ke Tang, and Xin Yao
Southern University of Science Technology, CHINA

***Abstract*—**Traditional solvers for tackling combinatorial optimization (CO) problems are usually designed by human experts. Recently, there has been a surge of interest in utilizing deep learning, especially deep reinforcement learning, to automatically learn effective solvers for CO. The resultant new paradigm is termed neural combinatorial optimization (NCO). However, the advantages and disadvantages of NCO relative to other approaches have not been empirically or theoretically well studied. This work presents a comprehensive comparative study of NCO solvers and alternative solvers. Specifically, taking the traveling salesman problem as the testbed problem, the performance of the solvers is assessed in five aspects, i.e., effectiveness, efficiency, stability, scalability, and generalization ability. Our results show that the solvers learned by NCO approaches, in general, still fall short of traditional solvers in nearly all these aspects. A potential benefit of NCO solvers would be their superior time and energy efficiency for small-size problem instances when sufficient training instances are available. Hopefully, this work would help with a better understanding of the strengths and weaknesses of NCO and provide a

Digital Object Identifier 10.1109/MCI.2023.3277768
Date of current version: 13 July 2023

Corresponding author: Ke Tang (e-mail: tangk3@sustech.edu.cn).

comprehensive evaluation protocol for further benchmarking NCO approaches in comparison to other approaches.

I. Introduction

Combinatorial optimization (CO) concerns optimizing an objective function by selecting a solution from a finite solution set, with the latter encoding constraints on the solution space. It has been involved in numerous real-world applications in logistics, supply chains, and energy [1]. From the perspective of computational complexity, many CO problems are NP-hard due to their discrete and nonconvex nature [2]. In recent decades, methods for solving CO problems have been extensively developed and can be broadly categorized into exact and approximate/heuristic/meta-heuristic methods [3]. The former methods are guaranteed to optimally solve CO problems but suffer from an exponential time complexity. In contrast, the latter methods seek to find good (but not necessarily optimal) solutions within reasonable computation time, i.e., they trade optimality for computational efficiency.

In general, most (if not all) of the above methods are manually designed. By analyzing the structure of the CO problem of interest, domain experts would leverage the algorithmic techniques that most effectively exploit this structure (e.g., proposed in the literature) and then continuously refine these methods (e.g., introducing new algorithmic techniques). Such a design process heavily depends on domain expertise and could be extremely expensive in terms of human time. For example, although the well-known traveling salesman problem (TSP) [4] has been studied for approximately 70 years, its methods [5], [6], [7], [8], [9] are still being actively and relentlessly updated.

Inspired by the success of deep learning (DL) in fields such as image classification [10], machine translation [11], and board games [12], recently there has been a surge of research interest in utilizing DL, especially deep reinforcement learning (DRL), to automatically learn effective methods for CO problems [13]. The resultant new paradigm is termed neural combinatorial optimization (NCO) [14], [15]. For the sake of clarity, henceforth, the optimization methods (either hand-engineered or automatically learned) are called solvers and the ways to design solvers are called design approaches. Compared to the traditional manual approach, NCO exhibits a significant paradigm shift in solver design. As illustrated in Figure 1, traditional solver design process is human-centered, while NCO is a learning-centered paradigm that develops a solver by training. The training process of NCO essentially calibrates the parameters of the solver (model). Although this approach induces a greater offline computational cost, the training process allows solver design to be conducted in an automated manner and thus involves much less human effort.[1]

Despite the appealing features NCO might bring, its advantages and disadvantages relative to other approaches have not been clearly specified. More specifically, although numerous computational experiments comparing NCO solvers with other solvers have been conducted in NCO works, they are generally nonconclusive for several reasons. First, it is often the case that the state-of-the-art traditional solvers are missing in the comparison, which would distort the conclusion and undermine the whole validation process. For example, the Google OR-Tools [16] is widely considered by NCO works [17], [18], [19], [20], [21] to be the baseline traditional solver for the vehicle routing problems (VRPs); however, it performs far worse than the state-of-the-art solvers for VRPs [22]. Second, for traditional solvers, their default configurations (parameter values) are used when comparing them with NCO solvers learned from training sets. Such an approach neglects the fact that, when a training set is available, the performance of traditional solvers could also be significantly enhanced by tuning their parameters [23], [24]. In practice, it is always desirable to make full use of the available technologies to achieve the best possible performance. In fact, with the help of the existing open-source algorithm configuration tools [24], [25], [26], the tuning processes of traditional solvers can be easily automated with little human effort involved. Third, the benchmark instances used in the comparative studies are often quite limited in terms of problem types and sizes, making it difficult to gain insights into how these approaches would perform on problem instances with different characteristics. For example, for TSP, the main testbed problem in NCO, most works have only reported results obtained on randomly generated instances with up to 100 nodes [18], [20], [27], [28], [29], [30]. In comparison, traditional TSP solvers are generally tested on problem instances collected from distinct applications, with up to tens of thousands of nodes [5], [6], [7], [8], [9].

To better understand the benefits and limitations of NCO, this work presents a more comprehensive empirical study. Specifically, TSP is employed as the testbed problem, since it is the originally oriented problem for many widely-used architectures in NCO and thus the conclusions drawn from it could have strong implications for other problems. Three recently developed NCO approaches and three state-of-the-art traditional TSP solvers are involved in the experiments. These solvers are compared on five problem types with node numbers ranging from 50 to 10000. The performance of the solvers is compared in five aspects that are critical in practice, i.e., effectiveness, efficiency, stability, scalability, and generalization ability. In particular, the energy efficiency (in terms of electric power consumption) of the solvers is also investigated, since energy consumption is being recognized as an important factor for solver selection if the applications of solvers continue to develop. To the best of our knowledge, this is the first comparative study of NCO approaches and traditional solvers on TSPs that 1) considers five different problem types, 2) involves problem instances with up to 10000 nodes, 3) includes tuned traditional solvers in the comparison, and 4) investigates five different performance aspects including the energy consumption of the solvers.

The presented comprehensive empirical study has led to several interesting findings. First, traditional solvers still significantly outperform NCO solvers in finding high-quality solutions regardless of problem types and sizes. In particular, current NCO

[1]It is noted that NCO still requires human time and expertise to carefully construct the training set, which should sufficiently represent the target use cases of the learned solver. However, this is not an easy task. This point will be further discussed in Section V.

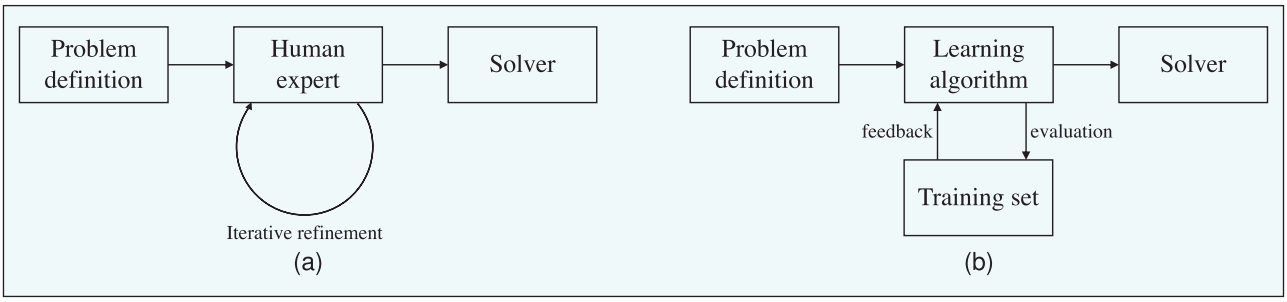

FIGURE 1 Illustrations of the two solver design paradigms. (a) Human-centered traditional paradigm. (b) Learning-centered NCO.

approaches are not adept at handling large-size problem instances and structural problem instances (e.g., clustered TSP instances). Second, a potential benefit of NCO solvers might be their efficiency (both in time and energy). For example, to achieve the same solution quality on small-size randomly generated problem instances, NCO solvers consume at most one-tenth of the resources consumed by traditional solvers. Third, when the training instances cannot sufficiently represent the target cases of the problem, both NCO solvers and tuned traditional solvers exhibit performance degradation, although the degradation is more dramatic for the former.

The remainder of the article is organized as follows. Section II briefly reviews the literature on NCO. Section III explains the design of the comparative study. Section IV presents the experimental results and analysis. Finally, concluding remarks are given in Section V.

II. Review of Neural Combinatorial Optimization

Before reviewing NCO, it is useful to first quickly recap typical CO solvers. In general, CO solvers include exact ones and approximate ones. Typical exact solvers are based on the branch-and-bound techniques that explore the solution space by branching into sub-problems and then filtering the set of possible solutions based on the upper and lower estimated bounds of the optimal solution. Typical approximate CO solvers are heuristics, which can be further roughly classified into constructive heuristics and improvement heuristics. The former incrementally builds a solution to a CO problem by adding one element at a time until a complete solution is obtained. In contrast, the latter improves upon a given solution by iteratively modifying it. The way of modifying a given solution is called a move operator. In recent decades, a lot of move operators have been proposed for different CO problems. For a comprehensive overview of CO, interested readers are referred to [1].

It is worth noting that the applications of neural networks to solve CO problems are actually not new. The earlier works [31] from the 80 s in the last century focused on using Hopfield neural networks (HNNs) to solve small-size TSP instances, which were later extended to other problems [32]. The main limitation of HNN-based approaches is that they need to use a separate network to solve each problem instance.

The term NCO refers to the series of works that utilize DL to learn a solver (model) to solve a set of different problem instances. According to the types of the learned solvers, the existing NCO approaches can be categorized into learning constructive heuristics (LCH), learning improvement heuristics (LIH), and learning hybrid solvers (LHS). As the names suggest, the solvers learned by LCH approaches and LIH approaches are constructive heuristics and improvement heuristics, respectively. Compared to traditional heuristics, their main differences are that the heuristic rules are no longer manually designed but are instead automatically learned. For example, the well-known greedy constructive heuristic for TSPs always selects the closest point for insertion, while LCH approaches learn a deep neural network (DNN) to score each point and finally select the point with the highest score for insertion. Compared to the manually designed greedy heuristic rule, the DNN model is trained with data and unnecessarily exhibits greedy behavior. Finally, LHS approaches seek to learn solvers that are hybrids of learning models and traditional solvers. The following sections will introduce these NCO approaches, mainly focusing on the key works. For a comprehensive survey of this area, interested readers are referred to [13], [33].

A. Learning Constructive Heuristics

1) Pointer Network-Based Approaches

As the seminal work, Vinyals et al. [27] introduced a sequence-to-sequence model, dubbed pointer network (Pr-Net), for solving TSPs defined on the two-dimensional plane. Specifically, Pr-Net is composed of an encoder and a decoder, and both of them are recurrent neural networks. Given a TSP instance, the encoder parses all the nodes in it and outputs an embedding (a real-valued vector) for each of them. Then, the decoder repeatedly uses an attention mechanism, which has been successfully applied to machine translation [11], to output a probability distribution over these previously encoded nodes, eventually obtaining a permutation over all the nodes, i.e., a solution to the input TSP instance. This approach allows the network to be used for problem instances with different sizes. However, Pr-Net is trained by supervised learning (SL) with precomputed near-optimal TSP solutions as labels. This could be a limiting factor, since in real-world applications such solutions of CO problems might be difficult to obtain. To overcome the above limitation, Bello et al. [28] proposed training Pr-Net with reinforcement learning (RL). In their implementation, the tour length of the partial TSP solution is used as the reward signal.

Another limitation of Pr-Net is that it treats the input as a sequence, while many CO problems have no natural internal

ordering. For example, the order of the nodes in a TSP instance is actually meaningless; instead, it should be better viewed as a set of nodes. To address this issue, Nazari et al. [17] replaced the recurrent neural network in the encoder of Pr-Net, which was supposed to capture the sequential information contained in the input, with a simple embedding layer that was invariant to the input sequence. In addition, the authors extended Pr-Net to solve VRPs, which differ from TSPs since VRPs involve dynamically changing properties (e.g., the demands of nodes) during the solution construction process. Specifically, the proposed model passes both the static properties (coordinates) and the dynamic properties (demands) of nodes and outputs an embedding for each of them. Then, at each decoding step, the decoder produces a probability distribution over all nodes while masking the serviced nodes and the nodes with demands that are larger than the remaining vehicle load.

2) Transformer-Based Approaches

In addition to the sequence-to-sequence model, the well-known transformer architecture [34] has also been applied to solve CO problems. In particular, transformer also follows the encoder-decoder framework but involves a so-called multi-head attention mechanism to extract deep features from the input. Such a mechanism has been used by Deudon et al. [29] to encode the nodes of TSP instances. Moreover, for sequentially decoding nodes, the authors of [29] proposed using only the last three decoding steps (i.e., the last three selected nodes) to obtain the reference vector for the attention mechanism, thus reducing the computational complexity. A similar model to that of [29] was implemented by Kool et al. [18]. Notably, the authors adjusted the model for many different CO problems including TSPs, prize collecting TSPs, VRPs, and orienteering problems, to accommodate their special characteristics. Additionally, the authors proposed an enhanced RL training procedure that used a simple rollout baseline and exhibited superior performance over [17], [29].

Based on the model proposed in [18], many subsequent works have improved it to achieve better solution quality or extended it to solve other VRP variants. For example, Peng et al. [35] adapted the model to re-encode the nodes during the solution construction process, obtaining better solution quality than the original model. A similar idea was implemented by Xin et al. [36], where the authors proposed changing the attention weights of the visited nodes in the encoder instead of completely re-encoding them. Another interesting work was completed by Li et al. [37], where the authors considered the multi-objective TSPs. They first decomposed the multi-objective problem into a series of single-objective subproblems and then used a Pr-Net to sequentially solve each sub-problem, where the network weights were shared among neighboring sub-problems. Finally, motivated by the fact that an optimal solution to a VRP instance, in general, has many different representations, Kwon et al. [15] introduced a modified RL training procedure to force diverse rollouts toward optimal solutions. The resultant approach, called POMO, is currently one of the strongest NCO approaches for learning constructive heuristics for TSPs and VRPs.

3) Graph Neural Network-Based Approaches

Another line of works leverages graph neural networks (GNNs) [38] to address the aforementioned issue of having an order-invariant input. Specifically, GNNs deal with graphs as inputs without considering the order of input sequence. Khalil et al. [39] introduced a GNN model for solving several graph CO problems including maximum cut problems, minimum vertex cover problems, and TSPs. The model, trained with RL, learns the embeddings of the nodes in the input problem instances, and then greedily selects nodes to construct a complete solution. It is also possible to integrate the node embeddings learned by GNNs into Pr-Net, as shown by Ma et al. [40]. Based on [39], subsequent works have extended GNNs to solve many other CO problems defined on graphs. For example, Li et al. [41] utilized GNNs to solve the maximal independent set problems and maximal clique problems. Unlike TSPs, for these problems the goal is not to find a permutation of nodes but a subset of nodes. Hence, instead of sequentially extending a solution, the authors used SL to train a graph convolutional network (GCN) to directly output an estimate of the probability of selecting each point and then utilized a guided tree search to construct a feasible solution based on these estimates. A similar work was brought forward by Joshi et al. [30], where the authors trained a GNN by SL to predict the probability of an edge being in the final TSP solution and then constructed a feasible tour by beam search.

4) Discussion

Due to their frameworks of sequentially encoding and decoding, Pr-Net-based approaches and transformer-based approaches are intrinsically suitable for handling permutation-based problems (e.g., TSPs and VRPs), where the orders of node selection form problem solutions. Among these two types of approaches, transformer-based approaches can achieve better performance mainly due to their advanced multi-head attention mechanism. In contrast, GNN-based approaches are suitable for handling CO problems defined on graphs and have no requirement regarding the sequential characteristics of the problems.

Overall, compared to LIH and LHS, LCH requires the least expert knowledge about the problems to be solved and therefore has the most potential to become a domain-independent solver design framework. However, with respect to obtaining high-quality solutions, current solvers learned by LCH approaches still perform worse than those learned by LIH approaches [21], [42] and LHS approaches [43].

B. Learning Improvement Heuristics

Unlike LCH approaches that learn models to sequentially extend a partial solution for a given problem instance, LIH approaches seek to learn a policy that manipulates local search operators to improve a given solution. Nonetheless, the encoder/decoder models used by LIH approaches are still similar to those used by LCH approaches.

In an early work, Chen and Tian [19] proposed learning two models to control the 2-opt operator, which is a

conventional move operator for VRPs. Specifically, the region-picking model selects a fragment of a solution to be improved and the rule-picking model selects a rewriting rule to be applied to the region. Both models are trained by RL and the solution is improved continuously until it converges.

Many subsequent works have improved upon [19] to achieve better solution quality. For example, Costa et al. [44] proposed learning separate embeddings for nodes and edges in the solution; Wu et al. [42] simplified the approach by learning only one model to select node pairs that are subject to the 2-opt move operator. Another notable work is that of Lu et al. [45]. Unlike previous approaches, the authors of [45] proposed learning a model to control several different move operators and applied a random permutation operator to the solution if the quality improvement could not reach the threshold. Finally, motivated by the circularity and symmetry of VRP solutions (i.e., cyclic sequences), Ma et al. [21] proposed a cyclic positional encoding mechanism to learn embeddings for positional features, which are independent of the node embeddings. The decoder and the employed move operators are similar to those of [42]. The resultant LIH approach, called DACT, has achieved superior performance on solving VRPs in comparison with other LCH approaches and LIH approaches.

Overall, compared to LCH, LIH integrates more expert knowledge about the problems (move operators) and can achieve better solution quality than the former. On the other hand, the application scope of LIH approaches is inevitably limited by the operators they integrate. For example, the above-mentioned approaches cannot be applied to CO problems without sequential characteristics (e.g., maximum cut problems and minimum vertex cover problems) because their move operators are inapplicable to these problems. Moreover, because the solvers learned by LIH approaches employ an iterative local search procedure, they need to consume more computation time than the solvers learned by LCH approaches [21].

C. Learning Hybrid Solvers

As aforementioned, LHS approaches seek to learn solvers that are hybrids of learning models and traditional solvers. It is worth mentioning that the integration of learning models (such as neural networks) into solvers is a long-standing research topic. For example, many studies have been conducted on integrating HNNs into evolutionary algorithms (EAs) [46], [47]. Below the research line of using DL and DRL to train such solvers is reviewed.

One early example is the neural large neighborhood search (NLNS) of Hottung and Tierney [48], which integrates a learning model into the well-known large neighborhood search (LNS) algorithm. Specifically, the use of extended/large neighborhood structures has widely proved to be effective for obtaining high-quality solutions to CO problems [49], [50]. LNS [51] is a typical algorithm framework that follows this idea. It explores the solution space by iteratively applying destroy-and-repair operators to a starting solution and has exhibited strong performance on a number of VRP variants. NLNS uses an attention-based model trained with RL as the repair operator for LNS. Later, Chen et al. [52] and Gao et al. [53] introduced two different variants of NLNS. The former trains a hierarchical recursive GCN as the destroy operator, while the latter uses an elementwise GNN with edge embedding as the destroy operator. Both approaches adopt a fixed repair operator that simply inserts the removed nodes into the solution according to the minimum cost principle.

In addition to LNS, another notable example is the Lin-Kernighan-Helsgaun (LKH) algorithm [5], which is widely recognized as a strong solver for TSPs. During the solution process, LKH iteratively searches for λ-opt moves based on a small candidate edge set to improve the existing solution. Zheng et al. [54] proposed training a policy that helps LKH select edges from the generated candidate set. However, the policy is trained for each instance instead of a set of instances. Later, Xin et al. [43] proposed training a GNN with SL to predict edge scores, based on which LKH can create the candidate edge set and transform edge distances to guide the search process. The resultant LHS approach, called NeuroLKH, has learned solvers that remarkably outperform the original LKH algorithm in obtaining high-quality solutions when solving TSPs and VRPs.

Compared to LCH and LIH, LHS integrates the most expert knowledge (traditional solvers) about the problems and can obtain the best solution quality [43]. However, since LHS relies on the existing solvers, its application scope is limited to the problems for which strong solvers exist. Moreover, a LHS approach is generally specifically tailored for a solver (e.g., NeuroLKH is tailored for LKH) and it is difficult to be extended to other solvers/problems.

III. Comparative Studies

This section first explains the design principle and the overall framework of the comparative study, then elaborates on the details, and finally summarizes the main differences between this study and the previous ones.

Specifically, the whole study is designed to simulate two typical scenarios that arise in practice when a practitioner is faced with a CO problem to solve. In the first scenario, one is aware of the target problem instances that the solver is expected to solve and can collect sufficient training instances to represent them. As an illustrative example, consider a delivery company that needs to solve TSP instances for the same city on a daily basis, with only slight travel time differences across the instances due to varying traffic conditions. In this example, one can use the accumulated instances to sufficiently represent the target use cases of the TSP solver. Suppose that the company expands its delivery business to another city, which differs from the first city in terms of their sizes, traffic conditions, and customer distributions. Then, the decision maker of the company faces the second scenario, in which the information of the target use cases of the solver is unavailable; thus, the decision maker expects the solver to handle problem instances with a broad range of problem characteristics.

From the perspective of computational study, the first scenario corresponds to the setting where the training instances and the testing instances have the same problem characteristics. NCO approaches are intrinsically appropriate for learning solvers in this scenario. On the other hand, traditional solvers can be directly applied to the testing instances or can first be tuned with the training instances and then tested. Furthermore, in this scenario practitioners are often concerned about the following aspects regarding the performance of the solvers.

1) **Effectiveness**: the extent to which the solver can solve the problem instances, generally measured by solution quality.
2) **Efficiency**: the computational resources (energy and computation time) consumed by the solver.
3) **Stability**: the extent to which the output of the solver is affected by its internal randomness.
4) **Scalability**: the problem sizes that the solver can handle. This is a natural performance consideration for traditional solvers. For NCO, it can be easily mixed up with generalization (see its definition below). More specifically, the scalability of an NCO approach refers to its ability to learn solvers as the problem size grows.

Unlike the first scenario, the second scenario corresponds to the experimental setting where the testing instances significantly differ from the training instances. In this scenario, practitioners expect the solvers to generalize well from training instances to unseen testing instances.

5) **Generalization**: how the learned solver would perform on instances that have different characteristics (e.g., problem sizes) from those of the training instances.

In the comparative study, NCO solvers and traditional solvers are evaluated in the above two scenarios. In particular, this work takes TSP as the testbed problem to elaborate on the design of the experiments. As a conventional CO problem, TSP has been studied for many years, and a number of strong traditional solvers have been developed for it [5], [6], [7], [8], [9]. More importantly, TSP has been the testbed problem for nearly all leading architectures in NCO [18], [20], [21], [27], [28], [36], [43]; thus, the most recently proposed NCO approaches that have achieved strong performance can be included in the experiments, and the conclusions drawn from this study may also have strong implications for other CO problems.

Since most NCO solvers are trained to handle the EUC-2D TSP instances, where the nodes are defined on a two-dimensional plane and the distances between two nodes are the same in both directions, this study also considers the EUC-2D TSP instances.

A. Overall Framework

The whole experiments aim to answer the following five research questions.

1) **Q1**: In the first scenario, how would the solvers perform on small-size problem instances?
2) **Q2**: In the first scenario, how would the solvers perform on medium/large-size problem instances?
3) **Q3**: In the second scenario, how would the solvers generalize over different problem types (i.e., characterized by node distributions)?
4) **Q4**: In the second scenario, how would the solvers generalize over different problem sizes?
5) **Q5**: In the second scenario, how would the solvers generalize over different problem types and sizes?

Specifically, the first two questions are concerned with the effectiveness, efficiency, stability, and scalability of the solvers in the first scenario where the training instances and the testing instances have the same problem characteristics. The other three questions are concerned with the generalization ability of the solvers in the second scenario, where the training instances and the testing instances have different problem characteristics.

Each of the above questions is investigated in a separate group of experiments, denoted as Exp_1/2/3/4/5. Note that throughout the experiments, training instances are only used for learning NCO solvers or tuning traditional solvers, and all the solvers are tested on the testing instances. The training/testing sets in each group of experiments, the compared methods, and the evaluation metrics are further elaborated below.

B. Benchmark Instances

Two different sources for obtaining TSP instances were considered: data generation and existing benchmark sets. Specifically, for data generation, the *portgen* generator which has been used to create testbeds for the 8-th DIMACS Implementation Challenge [4] and the *ClusteredNetwork* generator from the *netgen* R-package [55] were used.

1) The *portgen* generator generates a TSP instance (called a *rue* instance) by uniformly and randomly placing points on a two-dimensional plane.
2) The *ClusteredNetwork* generator generates a TSP instance (called a *clu* instance) by randomly placing points around different central points.

Three benchmark sets, i.e., *TSPlib*, *VLSI*, and *National*, were used:

1) *TSPlib* [56]: a widely used benchmark set of instances drawn from industrial applications and geographic problems featuring the locations of cities (nodes) on maps.
2) *VLSI:* a benchmark set of instances extracted from the very-large-scale integration design data of the Bonn Institute.
3) *National:* a benchmark set of instances extracted from the maps of different countries.[2]

For all the generated instances, Concorde [57], an exact TSP solver, was used to obtain their optimal solutions.[3] For the instances belonging to the existing benchmark sets, their optimal solutions or best-known solutions (in case the optimal solutions are unknown) were collected and used.

Based on the above data generation/collection procedure, the training/testing sets in each of the five groups of experiments were constructed as follows (also summarized in Table I):

[2]All three benchmark sets are available at http://www.math.uwaterloo.ca/tsp/data.
[3]Concorde is available at https://www.math.uwaterloo.ca/tsp/concorde.html.

TABLE I The training sets and the testing sets (separated by "|") in five groups of experiments. "Exist. Bench." refers to the testing set containing 30 instances selected from the existing benchmark sets.

EXPERIMENT GROUP	SCENARIO 1		SCENARIO 2		
	EXP_1	EXP_2	EXP_3	EXP_4	EXP_5
Training Set | Testing Set	*rue*-50 | *rue*-50 *clu*-50 | *clu*-50 *rue*-100 | *rue*-100 *clu*-100 | *clu*-100	*rue*-500 | *rue*-500 *clu*-500 | *clu*-500 *rue*-1000 | *rue*-1000 *clu*-1000 | *clu*-1000	*rue*-100, *mix*-100 | *clu*-100 *clu*-100, *mix*-100 | *rue*-100 *rue*-1000, *mix*-1000 | *clu*-1000 *clu*-1000, *mix*-1000 | *rue*-1000	*rue*-50 | *rue*-100 *clu*-50 | *clu*-100	*rue*-1000 | Exist. Bench. *clu*-1000 | Exist. Bench. *mix*-1000 | Exist. Bench.
Description	The training instances and the testing instances have the same problem characteristics		The training instances and the testing instances differ in either problem types, problem sizes, or both.		

1) **Exp_1**: Two problem sizes (50 and 100) and two problem types (*rue* and *clu*) were considered. Consequently, four combinations were produced, denoted as *rue*/*clu*-50/100. For each of them, following the common practice in NCO [18], [20], [21], one million training instances and 10000 testing instances were generated. For the *clu* instances, the number of clusters was set to $n/10$, where n was the problem size.
2) **Exp_2**: The whole procedure for constructing training/testing sets was exactly the same as that used in Exp_1, except that the considered problem sizes were 500/1000 and the testing set size was 1000. Besides, for the *clu* instances, the number of clusters was set to $n/100$.
3) **Exp_3**: Unlike Exp_1 and Exp_2, in this experiment, the testing instances differed from the training instances in problem types. Specifically, two problem sizes (100 and 1000) and two problem types (*rue* and *clu*) were considered. The solvers learned on the training set of *rue*-100/1000 instances would be tested on the testing set of *clu*-100/1000 instances, and vice versa. Moreover, in addition to the *rue* and *clu* training sets, another training set called *mix* was also used, which contained half *rue* instances and half *clu* instances.
4) **Exp_4**: In this experiment, the testing instances differed from the training instances in problem sizes. Specifically, two problem sizes (50 and 100) and two problem types (*rue* and *clu*) were considered. The solvers learned on the *rue*-50 training instances and the *clu*-50 training instances would be tested on the *rue*-100 testing instances and the *clu*-100 testing instances, respectively.
5) **Exp_5**: In this experiment, problem instances selected from the *TSPlib*, *VLSI*, and *National* benchmark sets were used as the testing instances. Specifically, 10 instances were selected from each of these three sets, with problem sizes distributed between 1000 and 10000. In addition, three training sets were considered in this experiment, i.e., *rue*-1000, *clu*-1000, and *mix*-1000.

TABLE II The competitors in the experiments.

METHOD	TYPE
POMO [20]	Learning Constructive Heuristics (LCH)
DACT [21]	Learning Improvement Heuristics (LIH)
NeuroLKH [43]	Learning Hybrid Solvers (LHS)
LKH [5]	Traditional Solver
EAX [7]	Traditional Solver
MAOS [58]	Traditional Solver
LKH (tuned)	Tuned Traditional Solver

C. Compared Methods

Table II lists all the competitors in the experiments. For each of the three types of NCO approaches, a recently proposed approach that has achieved strong performance was considered. Specifically, POMO [20], DACT [21], and NeuroLKH [43] were the considered approaches for LCH, LIH, and LHS, respectively. According to the results reported in [20] and [21], POMO and DACT could achieve their best performance with an extra instance augmentation mechanism; thus, these variants of POMO and DACT were also considered in the experiments. All the hyper-parameters of these approaches were set as reported in their original papers, except that in the experiments, their batch sizes were always tuned to fully utilize the GPU memory.

Regarding traditional solvers, except for the widely adopted LKH (version 3.0) [5] in NCO works, this study included two other meta-heuristic solvers, EAX [7] and MAOS [58], in the experiments.[4] EAX is a genetic algorithm equipped with a powerful edge assembly crossover. It has proved to outperform LKH in solving a broad range of TSP instances [7]. MAOS is a strong swarm intelligence-based TSP solver that does not contain any explicit local search heuristic. The parameters of LKH, EAX, and MAOS were kept as their default values in the experiments. Moreover, LKH poses many parameters whose values may significantly affect its performance; it is thus possible to tune these parameters on a training set to achieve better performance. Hence, in the experiments, the tuned variants of LKH obtained by using the general-purpose automatic algorithm configuration tool SMAC [24] were also considered. Generally, the computation time needed by SMAC to tune LKH was much shorter than that needed by NCO approaches to train their solvers.

[4]Concorde was not included in the comparison because it needs to run for prohibitively long periods of time to solve those very-large-size problem instances (e.g., larger than 5000). Besides, LKH, EAX, and MAOS could achieve solution quality very close to that of Concorde, while consuming much less computation time than the latter.

D. Evaluation Metrics

The testing results of the solvers are reported in terms of three metrics, i.e., optimum gap, computation time, and energy. For all three metrics, the smaller the results are, the better. Specifically, the optimum gap is defined as

$$(Q - Q^*)/Q^*,$$

where Q is the length of the tour found by the solver and Q^* is the optimal tour length.

The computation time of a solver is the time it takes to solve all the instances in the testing set. Note that NCO solvers would naturally benefit from running on massively parallel hardware architectures, i.e., GPUs, while in previous comparative studies [17], [18], [20], [21], [28] traditional solvers were generally run on CPUs using a single thread. To conduct fair comparisons, in the experiments, the traditional solvers and their tuned variants were also run on k CPU threads to solve k problem instances in parallel ($k = 32$ on our reference machine). Nevertheless, it is noted that the different programming languages adopted by NCO solvers and traditional solvers would also affect their runtime, and this cannot be avoided in our experiments. Specifically, NCO solvers are usually implemented with Python that mixes inefficient interpreted code with efficient DL libraries (e.g., those for utilizing GPUs). On the other hand, traditional solvers are typically implemented in highly efficient languages such as C/C++/java. Currently, how to avoid the influence of different programming languages when comparing NCO solvers and traditional solvers is still an open question.

Finally, the energy is the electric power consumed by a solver for solving all the instances in the testing set, which is a particularly useful metric in resource-limited cases such as embedded devices. In the experiments, the open-source PowerJoular tool was used to record the electric power consumed by the solvers.[5]

When testing the solvers, to prevent them from running for prohibitively long periods of time, the maximum runtime for solving a testing instance was set to 3600 seconds. If a solver consumed its time budget, it would be terminated immediately and the best solution found by it would be returned. Note that some tested solvers (e.g., EAX and LKH) involve randomized components. In the experiments, these solvers were applied on each testing instance for 10 runs. Then, the mean value, as well as the standard deviation of the optimum gaps over the 10 runs were recorded, which were further averaged over all the testing instances to obtain the average optimum gap and the average standard deviation on the whole testing set.

All the experiments were conducted on a server with an Intel Xeon Gold 6240 CPU (2.60 GHz, 24.75 MB of Cache) and an NVIDIA TITAN RTX GPU (24 GB of video memory) with 377 GB of RAM, running Ubuntu 18.04. The complete experimental results, benchmark instances, NCO solvers, traditional solvers and their tuned variants, and codes for training/tuning solvers are available at https://github.com/yzhang-gh/benchmarking-tsp.

[5]PowerJoular is available at https://github.com/joular/powerjoular.

E. Main Differences From Previous Comparative Studies

In general, the above established experimental protocol could be used as a standard protocol for benchmarking NCO approaches. More specifically, our comparative study differs from the studies presented in previous NCO works [17], [18], [20], [21], [28], [43] in the following aspects.

1) Regarding benchmark instances, in the NCO literature, it is common to use the *rue* type of instances as both training and testing instances, and some studies used *TSPlib* to assess the generalization ability of their learned solvers. Compared to them, this study used three more types of problem instances (*clu*, *National*, and *VLSI*) in the experiments, leading to testbed problems with much more diverse characteristics. Moreover, the considered problem sizes ranged from 50 to 10000, which were much larger than those in the previous NCO works.
2) Regarding traditional solvers, in addition to the LKH solver widely adopted by NCO works, this study included two other strong solvers (EAX and MAOS) in the comparison to fully represent the state-of-the-art TSP solvers. To assess the potential of traditional solvers, this study also considered tuning their parameters, which to the best of our knowledge has never been considered by the existing NCO works.
3) This study investigated five different performance aspects and introduced a new efficiency metric, i.e., electric power consumption, which could be particularly useful in energy-limited environments. Besides, to conduct fair comparisons in terms of time efficiency, all the NCO solvers and traditional solvers (and their tuned variants) were tested in the parallel mode to make full use of our reference machines. In comparison, previous comparative studies often tested traditional solvers on CPUs using a single thread.

IV. Experimental Results and Analysis

This section first presents the main findings drawn from the experiments and then analyzes the results of each group of experiments in detail.

A. Main Findings

Overall, four main findings can be obtained based on the experimental results.

First, for all the TSP problem sizes and types considered in the experiments, traditional solvers still significantly outperformed NCO solvers in finding high-quality solutions (Sections IV-B, IV-C, and IV-D). Among NCO solvers, the hybrid solvers trained by LHS approaches could find much better solutions than the learned constructive and improvement heuristics. In other words, the more expert knowledge that was integrated in an NCO solver, the better it solved the problems. Hence, it appears that the research status in this area has not yet reached those in domains such as vision, speech, and natural language processing, where DL can learn strong models from scratch.

Second, due to their simple solving strategy (i.e., sequentially constructing a solution) and massively parallel computing mode, a major potential benefit of NCO solvers (i.e., the constructive heuristics learned by LCH approaches) is their superior efficiency (in terms of both time and energy). In particular, on small-size randomly generated problem instances, the computational resources consumed by the learned heuristics were usually at most one-tenth of the resources consumed by traditional solvers (Section IV-B), where the latter were terminated once they achieved the same solution quality as that of the former.

Third, current LCH and LIH approaches are not suitable for handling large-size problem instances (Section IV-C) and structural problem instances (Sections IV-B and IV-C), e.g., the *clu* type of TSP instances.

Fourth, parameter tuning can significantly boost the performance of traditional solvers in terms of solution quality while maintaining efficiency (Section IV-D). However, when the training instances had different problem characteristics (problem types and sizes) from those of the testing instances, both NCO solvers and tuned traditional solvers exhibited performance degradation (Sections IV-D

TABLE III Testing results of Exp_1/2. For each metric, the best performance is indicated in gray. LKH* and EAX* refer to the variants of LKH and EAX, respectively, which were terminated once they achieved the same solution quality as that of POMO solver.

Exp_1: Small-Size Problem Instances

Method	*rue*-50			*clu*-50		
	Gap (%) ± std (%)	Time (s)	Energy (J)	Gap (%) ± std (%)	Time (s)	Energy (J)
POMO, no aug.	0.1185 ± 0.0000	2.57	290.83	0.1353 ± 0.0000	2.58	292.14
POMO, ×8 aug.	0.0228 ± 0.0000	16.98	3361.87	0.0213 ± 0.0000	17.04	4193.39
DACT	0.0167 ± 0.0291	1991.49	402635.40	0.1770 ± 0.1117	1921.99	393994.52
DACT, ×4 aug.	0.0006 ± 0.0013	8534.42	1742933.29	0.0576 ± 0.0390	8735.50	1676140.80
NeuroLKH	0.0003 ± 0.0003	34.22	4006.11	0.0004 ± 0.0007	131.92	12698.78
MAOS	0.0000 ± 0.0000	357.33	37219.44	0.0000 ± 0.0000	350.84	36387.72
LKH	0.0035 ± 0.0035	291.22	10458.70	0.0022 ± 0.0018	257.95	9512.24
EAX	0.0000 ± 0.0000	343.79	15089.39	0.0000 ± 0.0000	321.06	12541.47
LKH*	Same as POMO	259.17	9551.16	Same as POMO	207.11	6228.23
EAX*		273.95	10248.02		267.95	10165.34
Method	*rue*-100			*clu*-100		
	Gap (%) ± std (%)	Time (s)	Energy (J)	Gap (%) ± std (%)	Time (s)	Energy (J)
POMO, no aug.	0.3646 ± 0.0000	12.59	2588.90	0.4318 ± 0.0000	12.83	2675.18
POMO, ×8 aug.	0.1278 ± 0.0000	87.73	25873.38	0.1405 ± 0.0000	93.08	27299.18
DACT	0.6596 ± 0.5216	6141.72	1269009.13	1.2220 ± 0.4773	6517.2110	1390740.24
DACT, ×4 aug.	-			-		
NeuroLKH	0.0004 ± 0.0005	74.90	9819.77	0.0021 ± 0.0031	309.75	29397.05
MAOS	0.0000 ± 0.0000	451.94	55214.54	0.0000 ± 0.0000	447.51	54716.25
LKH	0.0044 ± 0.0048	313.73	12868.38	0.0048 ± 0.0040	340.58	14264.64
EAX	0.0000 ± 0.0000	598.34	35145.37	0.0000 ± 0.0000	561.41	27893.49
LKH*	Same as POMO	262.99	10707.67	Same as POMO	294.30	12140.67
EAX*		366.99	22638.50		340.67	13422.11

Exp_2: Medium/Large-Size Problem Instances

Method	*rue*-500			*clu*-500		
	Gap (‰) ± std (‰)	Time (s)	Energy (J)	Gap (‰) ± std (‰)	Time (s)	Energy (J)
NeuroLKH	0.0273 ± 0.0441	242.71	21455.69	1.8080 ± 2.5469	2695.06	171826.14
MAOS	0.0870 ± 0.1591	133.08	13720.29	0.1502 ± 0.1674	111.20	11728.91
LKH	0.4356 ± 0.5084	143.36	17077.63	4.2779 ± 3.6307	313.80	35213.13
LKH (tuned)	0.0086 ± 0.0165	162.63	19087.78	0.0345 ± 0.0398	279.80	32685.72
EAX	0.0140 ± 0.0291	269.41	32370.64	0.0006 ± 0.0012	209.09	24456.69
Method	*rue*-1000			*clu*-1000		
	Gap (‰) ± std (‰)	Time (s)	Energy (J)	Gap (‰) ± std (‰)	Time (s)	Energy (J)
NeuroLKH	0.0417 ± 0.0510	728.41	58569.96	1.8599 ± 2.2236	5094.77	334771.95
MAOS	0.1263 ± 0.1561	380.32	37621.01	0.2096 ± 0.1863	272.97	27621.76
LKH	0.3620 ± 0.3624	413.43	48867.49	1.8761 ± 1.2798	859.39	100514.45
LKH (tuned)	0.1732 ± 0.1770	406.16	52252.97	0.0460 ± 0.0537	522.41	66511.40
EAX	0.0182 ± 0.0242	620.30	75168.05	0.0086 ± 0.0122	631.68	75222.30

and IV-E), and NCO solvers suffered from far more severe performance degradation.

B. Exp_1: Small-Size Testing Instances With the Same Problem Characteristics as Training Instances

The testing results of Exp_1 in terms of average optimum gap (Gap), standard deviation (std), total computation time, and energy are reported in Table III. In Exp_1, for DACT with the instance augmentation mechanism (denoted by aug.), its results on the *rue*-100 and *clu*-100 testing instances are missing because it ran for prohibitively long periods of time to solve these instances. Besides, the tuned variant of LKH, i.e., LKH (tuned), was not included in Exp_1 because the original LKH has already achieved nearly optimal solution quality. In addition, the medians and variance of the optimum gaps across all the testing instances are visualized by box plots in Figure 2. For brevity, the name of the NCO approach is used to denote the solvers learned by it.

The first observation from these results is that traditional solvers still achieved much better solution quality than the learned solvers. For example, EAX and MAOS could notably solve all the testing instances to optimality. Among all the randomized solvers, i.e., DACT, NeuroLKH, LKH, EAX, and MAOS, EAX and MAOS also exhibited the best stability. They achieved the smallest standard deviation over 10 repeated runs.

The second observation is that, after EAX and MAOS, NeuroLKH was the third best-performing solver. Compared to LKH, NeuroLKH reduced the average optimum gap by one order of magnitude on three out of the four testing sets. Based on Figure 2, one can also observe that NeuroLKH achieved more stable performance than LKH across the testing instances. Compared to the other two NCO solvers POMO and DACT, the performance advantages of NeuroLKH in terms of solution quality were much more significant. In general, NeuroLKH could reduce the average optimum gaps by at least two orders of magnitude on all four testing sets. Although the performance of POMO and DACT could be improved when equipped with the instance augmentation mechanism, they still performed worse than NeuroLKH.

The third observation is that NCO solvers could generally achieve better solution quality on the *rue* instances than on the *clu* instances. For example, the average optimum gap achieved by POMO on the *clu*-50 testing instances was 14.18% greater than that obtained on the *rue*-50 testing instances, and the corresponding numbers for DACT and NeuroLKH were 9.60 times and 33.33%, respectively. Moreover, as the problem size grows, such performance gaps became larger. These results show that current NCO approaches are less adept at learning solvers for structural problem instances (i.e., clustered TSP instances) than for uniformly and randomly generated instances, indicating that the learning models adopted by them may have limitations in handling structural data. This could be an important direction for improving NCO approaches.

The fourth observation is that in Figure 2, as the problem size grows, the performance of POMO and DACT significantly deteriorated, while the performance of NeuroLKH was still stable. These results indicate that currently the scalability of LCH approaches and LIH approaches is still quite limited.

The last observation is that regarding efficiency, POMO exhibited excellent performance in terms of both runtime and energy. Notably, it usually consumed at most one-tenth of the resources consumed by other solvers, which could be very useful in resource-limited environments. This is also true when EAX and LKH were terminated at the solution quality achieved by POMO solver (marked by LKH* and EAX* in

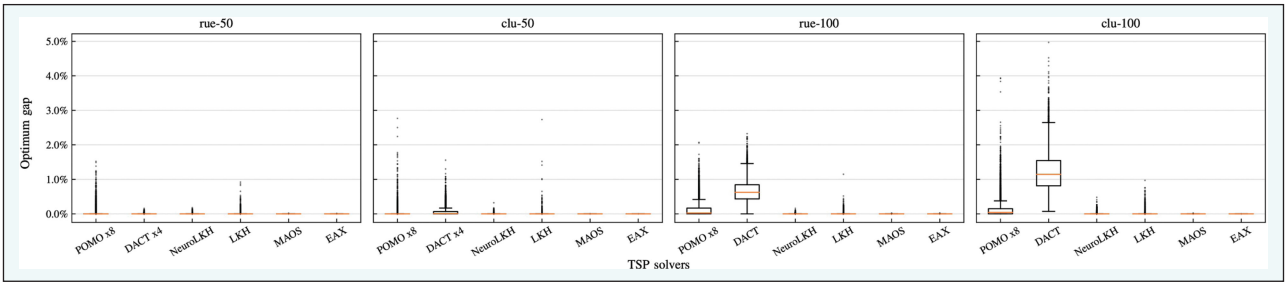

FIGURE 2 Visual comparison in box plots of the optimum gaps achieved by the tested solvers in Exp_1.

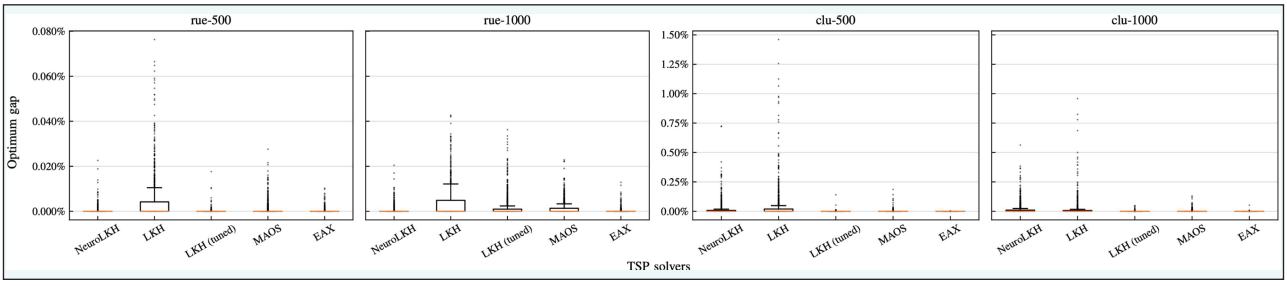

FIGURE 3 Visual comparison in box plots of the optimum gaps achieved by the tested solvers in Exp_2.

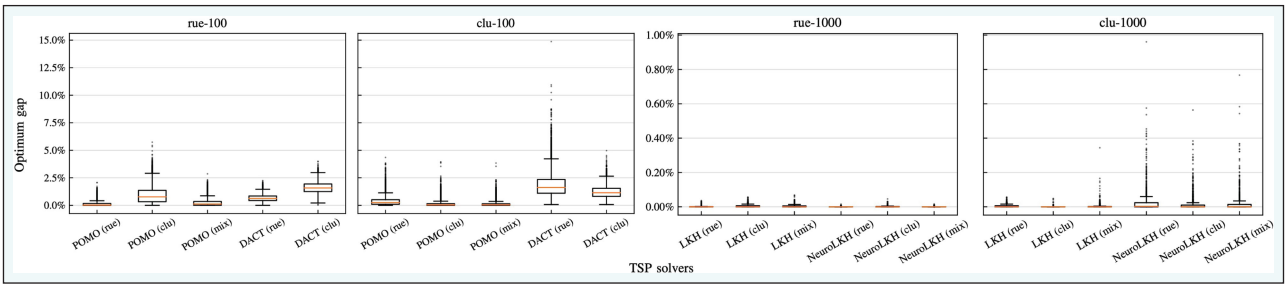

FIGURE 4 Visual comparison in box plots of the optimum gaps achieved by the tested solvers in Exp_3. Each learned/tuned solver is marked with the corresponding problem type of the training instances.

Table III). On the other hand, considering the poor scalability of POMO, its high efficiency was still limited to small-size problem instances. It is also found that another NCO approach, DACT, performed poorly in terms of efficiency, especially when equipped with the instance augmentation mechanism. Finally, NeuroLKH could improve the efficiency of the original LKH in both runtime and energy, and in general, the efficiency of EAX was slightly worse than the LKH-family solvers.

C. Exp_2: Medium/Large-Size Testing Instances With the Same Problem Characteristics as Training Instances

Similar to Exp_1, the testing results of Exp_2 are reported in Table III and illustrated in Figure 3. The main difference between Exp_1 and Exp_2 is that the latter considered much larger problem sizes. In Exp_2, POMO and DACT were not tested due to their poor scalability.

The first observation from these results is that overall, EAX is still the best-performing solver in terms of solution quality. Nevertheless, the tuned variant of LKH outperformed EAX on the *rue*-500 testing instances. Moreover, it outperformed MAOS on three out of the four testing sets, i.e., *rue*/*clu*-500 and *rue*-1000, while the original LKH fell behind MAOS on all four testing sets. Compared to the original LKH, the tuned variant of LKH could reduce the average optimum gaps by two orders of magnitude on three testing sets and by at least 50% on the remaining set. Based on Figure 3, it can also be observed that the tuned variant of LKH performed much more stably across the testing instances than LKH. It is worth mentioning that such performance improvement did not come at the cost of degraded efficiency. Overall, the tuned variant of LKH and the original LKH performed competitively in terms of time efficiency and energy efficiency. Such results indicate that traditional solvers can largely benefit from parameter tuning and this should be utilized when a sufficient training set is available.

The second observation is that although NeuroLKH could also achieve better solution quality than LKH, the former consumed much more computation time and energy than the latter. This is particularly evident on the *clu* testing instances. Such phenomenon once again implies that the current NCO approaches may have limitations on handling structural data. Taking a closer look at Figure 3, one can observe that on the *clu* testing instances, for LKH, parameter tuning could achieve greater performance improvement than NCO (i.e., NeuroLKH). This may be because parameter tuning can change the behaviors of LKH to a greater extent than that of NeuroLKH (which only modifies the candidate edge set in LKH), eventually leading to better fitting to the specific instance distribution. Such results also suggest an important future research direction of combining parameter tuning and NCO to achieve more comprehensive control over the behaviors of traditional solvers.

D. Exp_3: Testing Instances With Different Node Distributions From Training Instances

The medians and variance of the optimum gaps across all the testing instances in Exp_3 are illustrated by box plots in Figure 4. Note that DACT was unable to converge on the *mix* training set and thus the corresponding testing performance is not reported. Recalling that Exp_3 was designed to assess the generalization ability of the learned/tuned solvers over different problem types, the first observation from Figure 4 is that when applying a learned solver on the testing instances belonging to different types from those of the training instances, the solver's performance degraded. For example, on the *rue*-100 testing instances, the POMO solver trained on the *rue*-100 training instances performed better than the one trained on the *clu*-100 training instances, and the results were the opposite of the *clu*-100 testing instances. Based on the last two plots in Figure 4, one can observe that this was also true for NeuroLKH and the tuned variants of LKH.

The second observation is that when a *mix* training set was used, the learned/tuned solver still could not obtain the best possible performance. For example, on the *rue*-100 testing instances, the POMO solver trained on the *mix*-100 instances obtained an average optimum gap of 0.2420%, which is better than the one trained on the *clu*-100 instances (0.9234%) but still worse than the one trained on the *rue*-100 instances (0.1278%). These results indicate that the adopted learning models may not have sufficient capacity to simultaneously handle different problem types, which could be a direction for improving NCO approaches.

E. Exp_4: Testing Instances With Different Problem Sizes From Training Instances

Table IV presents the testing results of Exp_4 in terms of average optimum gap. Recalling that Exp_4 was designed to assess the generalization ability of the learned solvers over problem

TABLE IV Testing results of Exp_4. Each learned solver is marked with the corresponding training set.

METHOD (TRAINING SET)	*rue*-100	METHOD (TRAINING SET)	*clu*-100
	GAP (%) ± STD (%)		GAP (%) ± STD (%)
POMO (*rue*-50)	0.6703 ± 0.0000	POMO (*clu*-50)	0.6829 ± 0.0000
POMO (*rue*-100)	0.1278 ± 0.0000	POMO (*clu*-100)	0.1405 ± 0.0000
DACT (*rue*-50)	27.5437 ± 31.4449	DACT (*clu*-50)	21.4630 ± 5.3292
DACT (*rue*-100)	0.6596 ± 0.5216	DACT (*clu*-100)	1.2220 ± 0.4773

sizes, in Table IV each learned solver is marked with the corresponding problem sizes of its training instances.

The main observation is that when applying the solvers learned by POMO and DACT to the testing instances with larger sizes than the training instances, the performance of the solvers seriously degraded. For example, on the *rue*-100 testing instances, the DACT solver trained on the *rue*-50 training instances could only obtain an average optimum gap of 27.53%, which is generally an unacceptable level of solution quality for TSP. Such results demonstrate that although the learning models adopted by POMO and DACT can process variable-length inputs, it does not mean that the solvers trained by them can naturally generalize to larger problem sizes.

F. Exp_5: Testing Instances With Different Node Distributions and Problem Sizes From Training Instances

Recall that Exp_5 was designed to assess the ability of the learned/tuned solvers to generalize from generated instances to real-world instances, where the latter differed from the former in both problem sizes and problem types. Specifically, only the LKH-family solvers and EAX were tested in Exp_5 due to the large problem sizes. Three training sets were used in Exp_5, i.e., *rue*-1000, *clu*-1000, and *mix*-1000. Based on each training set, a tuned variant of LKH and a NeuroLKH solver were obtained. Then, for each testing instance, each solver was applied for 10 runs. Table V presents the testing results in terms of the number of times where the optimal solution was successfully found among the 10 runs, average optimum gap, and average computation time.

The first observation from Table V is that the best-performing solver is EAX. It achieved the highest number of successes on 22 out of the 30 testing instances, which is far more than that of the second best-performing solver. In particular, on the testing instances belonging to the *National* benchmark set, the performance gap between the LKH family solvers and EAX is significant, indicating that LKH may be intrinsically limited in solving this type of instances.

The second observation is that among all the LKH-family solvers, the solver learned by NeuroLKH on the *mix*-1000 training instances succeeded more times than the tuned variants of LKH and the original LKH. This may be because the mixed training set could cover more cases of possible TSP instances than the pure *rue* or *clu* training sets, finally leading to better generalization. On the other hand, based on the same training set, the NeuroLKH solver consistently performed better than the tuned variant of LKH. For example, the variant of LKH tuned on the *mix*-1000 training instances obtained fewer successes than the solver learned by NeuroLKH on the *mix*-1000 training instances, and the former was actually the one among all the LKH-family solvers that achieved the fewest successes. These results indicate that the parameter tuning process of LKH may cause the solver to be more easily overfitted to the training instances than the NCO approach NeuroLKH.

In summary, for real-world TSP instances that are unknown in advance, EAX is an appropriate default solver to use; to learn/train a TSP solver for these instances, NeuroLKH seems to be a better option than parameter tuning.

G. Learning Curves of NCO Solvers

It is meaningful to investigate the training phases of NCO solvers, since they have a significant impact on the solvers' performance. Figure 5 illustrates the learning curves of POMO solvers for *rue*/*clu*-50/100. After each training epoch, the POMO solver was evaluated on a validation set of 10000 problem instances; then, the average tour length of the obtained TSP solutions is plotted. Moreover, the average tour length of the optimal solutions and the GPU hours consumed for training POMO solvers are also illustrated in Figure 5. Figure 6 illustrates the learning curves of NeuroLKH for *rue*/*clu*-500. Note that for NeuroLKH, the trained model was used to generate a candidate edge set for LKH, not to directly solve the problem instances; thus, in Figure 6 the training loss is plotted.

From these results, one could make three observations. First, the validation performance of POMO solvers gradually improved as the training epochs increased and eventually sufficiently converged. Second, as the problem size increased from 50 to 100, the learning curves of POMO solvers converged more slowly, and the final optimum gaps became larger. This echoes the previous finding that the learning capability of POMO solvers is not sufficient for handling large-size problems. Finally, the training time of NCO solvers could vary from several GPU hours to several GPU days, but it is generally acceptable.

V. Conclusion

The applications of neural networks to solve CO problems have been studied for decades (starting from HNN-based works [31], [32]), and recently a subfield known as neural combinatorial optimization (NCO) has emerged rapidly. This work highlighted several issues exhibited by the comparative studies in the existing NCO works and presented an in-depth

TABLE V **Testing results of Exp_5. Each cell contains three values, i.e., number of successes, average optimum gap (%), and computation time (s). For each testing instance, the highest number of successes is indicated in gray, and the highest number of successes achieved among all LKH variants (including NeuroLKH and the tuned variants of LKH) is boxed.**

	Problem	LKH (default)	LKH (*rue*)	LKH (*clu*)	LKH (*mix*)	NeuroLKH (*rue*)	NeuroLKH (*clu*)	NeuroLKH (*mix*)	EAX
TSPlib	d1655	10/10, 0.0000 7.65s	9/10, 0.0161 15.18s	8/10, 0.0322 11.32s	5/10, 0.0805 13.51s	6/10, 0.0006 106.75s	7/10, 0.0483 83.18s	10/10, 0.0000 22.51s	10/10, 0.0000 27.94s
	fl3795	0/10, 81.0510 3621.86s	9/10, 0.0348 87.51s	10/10, 0.0000 10.48s	6/10, 1.8073 89.72s	0/10, 239.956 3609.97s	0/10, 186.744 3593.94s	0/10, 275.928 3582.66s	1/10, 15.3622 3246.70s
	fnl4461	8/10, 0.0548 107.55s	1/10, 1.0462 431.85s	0/10, 2.3882 341.85s	3/10, 0.6135 248.90s	10/10, 0.0000 72.61s	10/10, 0.0000 44.23s	10/10, 0.0000 36.10s	8/10, 0.0438 1135.55s
	pcb3038	6/10, 0.4575 110.66s	0/10, 1.4380 225.28s	0/10, 2.1424 142.03s	1/10, 1.4234 121.23s	9/10, 0.0363 110.28s	10/10, 0.0000 53.85s	9/10, 0.0363 137.30s	10/10, 0.0000 123.71s
	pla7397	10/10, 0.0000 598.89s	3/10, 0.5333 2473.04s	9/10, 0.0389 956.07s	5/10, 1.5073 1146.88s	0/10, 5.4551 3438.43s	1/10, 5.1922 3483.71s	5/10, 1.4884 2229.15s	0/10, 0.3647 3600.00s
	pr2392	10/10, 0.0000 2.56s	6/10, 0.0741 84.41s	1/10, 0.4471 74.18s	6/10, 0.2460 44.23s	10/10, 0.0000 4.84s	10/10, 0.0000 5.27s	10/10, 0.0000 2.46s	10/10, 0.0000 55.39s
	rl1889	0/10, 2.7675 161.49s	3/10, 0.1580 56.52s	6/10, 0.1074 26.06s	4/10, 3.3330 23.64s	1/10, 3.8037 147.02s	6/10, 1.5638 80.23s	2/10, 2.4926 208.98s	10/10, 0.0000 33.22s
	rl5934	1/10, 2.4027 615.60s	7/10, 0.0198 606.66s	5/10, 0.0252 469.67s	0/10, 4.2892 528.45s	5/10, 1.8883 654.52s	8/10, 0.9747 675.75s	10/10, 0.0000 405.83s	5/10, 0.8722 1939.15s
	u1817	1/10, 6.4160 130.73s	2/10, 4.7377 59.46s	1/10, 8.8110 38.49s	0/10, 0.9963 26.33s	2/10, 4.6503 210.41s	2/10, 5.5419 275.88s	1/10, 7.5523 273.12s	8/10, 1.4161 747.99s
	u2319	10/10, 0.0000 0.69s	10/10, 0.0000 0.07s	10/10, 0.0000 0.05s	10/10, 0.0000 0.03s	10/10, 0.0000 1.38s	10/10, 0.0000 1.37s	10/10, 0.0000 1.37s	1/10, 3.0309 3250.97s
VLSI	icw1483	10/10, 0.0000 9.39s	10/10, 0.0000 0.34s	10/10, 0.0000 0.47s	10/10, 0.0000 3.28s	10/10, 0.0000 3.59s	10/10, 0.0000 2.05s	10/10, 0.0000 7.17s	10/10, 0.0000 23.76s
	dcc1911	1/10, 2.0325 193.22s	3/10, 1.0944 69.81s	2/10, 2.0325 45.91s	2/10, 1.2508 35.51s	4/10, 0.9381 227.78s	1/10, 1.5635 311.04s	7/10, 0.4690 112.02s	9/10, 0.1563 400.82s
	xpr2308	4/10, 1.2467 141.41s	6/10, 0.8311 86.09s	5/10, 1.2467 61.84s	4/10, 0.8311 41.00s	9/10, 0.2770 151.17s	9/10, 0.2770 146.09s	10/10, 0.0000 103.27s	9/10, 0.1385 413.77s
	irw2802	3/10, 1.8996 198.79s	9/10, 0.3562 70.05s	8/10, 0.4749 41.85s	1/10, 2.0183 76.95s	8/10, 0.2374 164.17s	10/10, 0.0000 59.08s	10/10, 0.0000 70.69s	10/10, 0.0000 69.36s
	lta3140	8/10, 0.2102 221.66s	1/10, 1.7863 247.91s	1/10, 5.0436 140.03s	4/10, 1.5761 104.04s	7/10, 0.4203 417.62s	10/10, 0.0000 186.98s	9/10, 0.1051 167.16s	10/10, 0.0000 94.62s
	ltb3729	4/10, 1.0151 785.12s	3/10, 0.8460 348.86s	3/10, 1.6919 206.57s	2/10, 2.5379 159.37s	2/10, 1.3535 716.95s	9/10, 0.0846 718.67s	10/10, 0.0000 203.96s	10/10, 0.0000 125.47s
	bgb4355	2/10, 3.4583 617.08s	0/10, 6.9166 556.71s	1/10, 8.1742 297.43s	1/10, 6.0520 210.62s	2/10, 2.5937 1200.94s	2/10, 3.7727 1595.97s	8/10, 0.9432 556.52s	10/10, 0.0000 169.41s
	xqd4966	9/10, 0.0653 929.77s	10/10, 0.0000 103.56s	7/10, 0.1959 366.47s	6/10, 0.2612 235.34s	0/10, 9.3262 3553.00s	2/10, 1.6323 3113.29s	1/10, 1.3058 3284.33s	10/10, 0.0000 208.28s
	fea5557	6/10, 0.5827 490.28s	0/10, 4.8559 951.59s	0/10, 8.3522 503.51s	1/10, 5.3739 386.20s	0/10, 4.2085 1630.28s	10/10, 0.0000 258.24s	4/10, 0.7770 1754.45s	10/10, 0.0000 225.14s
	xsc6880	0/10, 3.2505 1831.57s	0/10, 2.5540 1695.90s	0/10, 5.2008 952.99s	0/10, 5.8509 740.00s	0/10, 3.1577 2777.26s	0/10, 2.2754 3427.23s	3/10, 1.2538 2684.66s	7/10, 0.2322 1386.64s
National	rw1621	0/10, 7.4469 2193.96s	10/10, 0.0000 31.75s	7/10, 0.2303 49.74s	0/10, 8.7129 72.18s	3/10, 0.8829 2292.70s	6/10, 0.5374 1331.73s	2/10, 1.5354 1977.19s	10/10, 0.0000 27.88s
	mu1979	7/10, 0.0691 125.62s	9/10, 0.0230 35.92s	8/10, 0.0460 20.98s	1/10, 1.3005 49.12s	0/10, 9.6463 1677.92s	0/10, 2.2913 180.62s	0/10, 0.9908 1724.79s	10/10, 0.0000 51.38s
	nu3496	0/10, 5.0660 3583.15s	0/10, 1.2795 1213.39s	0/10, 4.9827 360.71s	0/10, 6.3871 340.17s	0/10, 7.6041 3585.49s	3/10, 1.4147 3135.37s	3/10, 1.4771 2983.63s	10/10, 0.0000 133.82s
	ca4663	0/10, 0.1372 500.99s	9/10, 0.0093 155.47s	9/10, 0.0031 185.75s	3/10, 0.0279 323.63s	0/10, 1.5483 470.20s	0/10, 5.8908 509.78s	0/10, 3.5340 569.64s	10/10, 0.0000 381.83s
	tz6117	2/10, 0.4510 3333.54s	0/10, 1.3326 1322.48s	0/10, 2.2953 783.38s	0/10, 1.8520 676.98s	0/10, 6.4578 3552.03s	1/10, 0.2913 3475.44s	1/10, 0.1976 3490.76s	4/10, 0.1014 2361.93s
	eg7146	0/10, 9.5392 2093.50s	0/10, 2.3436 1611.97s	0/10, 7.1177 984.30s	0/10, 2.0949 893.78s	0/10, 0.3695 2053.23s	0/10, 6.0735 2309.80s	0/10, 0.6173 2965.75s	9/10, 0.0174 1204.65s
	pm8079	0/10, 4.3794 3471.67s	0/10, 2.1331 3510.69s	0/10, 2.9777 2507.03s	0/10, 3.2998 3215.28s	0/10, 1.3700 3481.58s	0/10, 6.1817 3469.07s	0/10, 2.5328 3474.12s	0/10, 5.0498 3600.00s
	ei8246	3/10, 0.3056 1426.69s	0/10, 1.9256 2152.78s	0/10, 4.4090 1375.38s	0/10, 2.0177 1344.62s	1/10, 0.5481 3133.45s	8/10, 0.0728 1545.12s	8/10, 0.1164 1808.24s	5/10, 0.0776 2418.63s
	ar9152	0/10, 3.6073 3419.22s	0/10, 6.9052 3486.53s	0/10, 0.8827 3108.36s	0/10, 0.2701 3303.10s	0/10, 0.7599 3416.96s	0/10, 7.5668 3420.46s	0/10, 3.3469 3416.12s	8/10, 0.0287 1585.88s
	kz9976	2/10, 0.7741 2144.65s	1/10, 1.3570 3304.81s	0/10, 3.0013 2299.72s	0/10, 1.4776 2223.27s	0/10, 7.9295 3037.43s	0/10, 1.2817 3372.06s	0/10, 1.4879 3375.52s	9/10, 0.0047 1365.65s

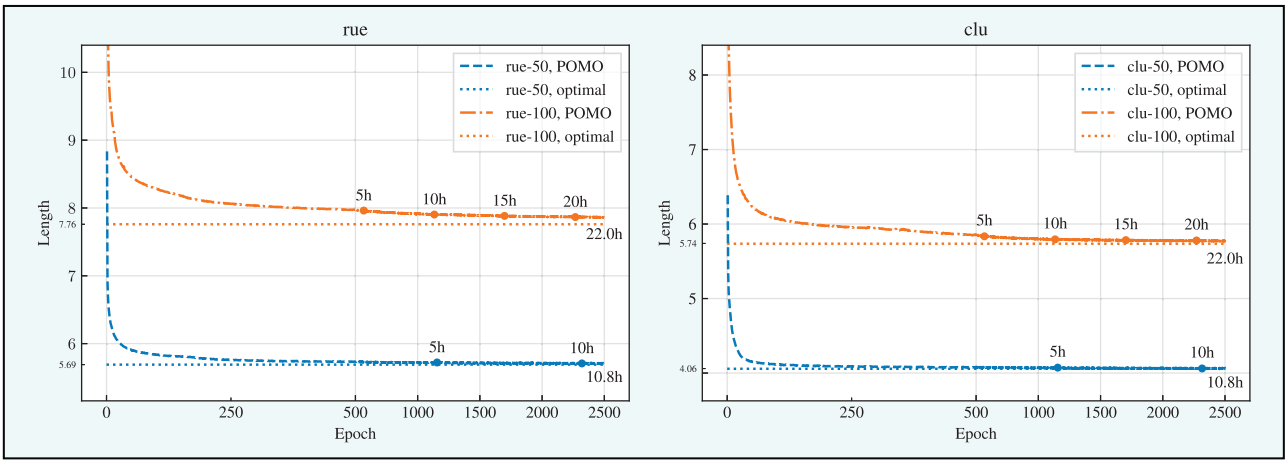

FIGURE 5 Learning curves of POMO solvers for *rue*-50/100 and *clu*-50/100.

comparative study of traditional solvers and NCO solvers on TSPs. An evaluation protocol driven by five research questions was established, which could be used as a basis for benchmarking NCO approaches against others on more CO problems. Specifically, two practical scenarios, categorized by whether one could collect sufficient training instances to represent the target cases of the problem, were considered. Then, the performance of the solvers was compared in terms of five critical aspects in these scenarios, i.e., effectiveness, efficiency, stability, scalability, and generalization ability. Five different problem types with node numbers ranging from 50 to 10000 were used as the benchmark instances in the experiments.

Based on the experimental results, it is found that, in general, NCO solvers were still dominated by traditional solvers in nearly all performance aspects. A potential benefit of NCO solvers might be their high efficiency (in terms of both time and energy) on small-size problem instances. It is also found that, for NCO approaches, a crucial assumption is that the training instances should sufficiently represent the target cases of the problem; otherwise, the trained solvers would exhibit severe performance degradation on the testing instances.

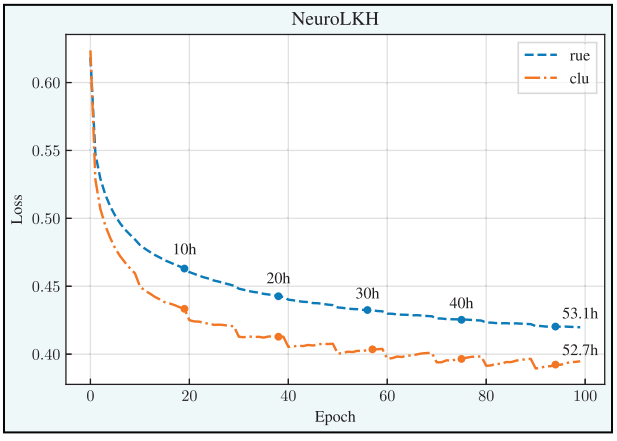

FIGURE 6 Learning curves of NeuroLKH for *rue*-500 and *clu*-500 in terms of training loss. The GPU hours consumed for training are also illustrated.

However, in many real-world applications, one can only collect a limited number of problem instances [59], [60], or the accumulated instances are outdated and cannot effectively reflect the current properties of the problem [61], [62], [63]. In these cases, collecting a good training instance set can take a significant amount of time and may even be impossible, which might reduce the potential advantage of NCO approaches.

As shown in the experiments, NCO faces several challenges that need to be dealt with in the future; several potential research directions are suggested.

1) Development of novel architectures or training algorithms to better handle structural problem instances.
2) Enhancement of current NCO approaches to learn solvers that can perform well on large-size problem instances and multiple (not one) problem types.
3) Hybridization of parameter tuning and NCO to achieve more comprehensive control over the behaviors of traditional solvers, hopefully leading to even better performance.

Finally, it is worth mentioning that the merits of a CO solver can always be considered from two different perspectives. The first is its strength, i.e., how well it can solve a particular CO problem, which is exactly the perspective adopted by this work. The second is the generality, i.e., how many different CO problems it can be used to solve. Recent studies have extended NCO with unified DNN models to many different CO problems [39], [41], where there might be no specialized solvers such as LKH and EAX. Hence, it seems that NCO is a potential alternative for general-purpose CO solvers. A systematic evaluation study of the generality of NCO approaches has the potential for future research.

Acknowledgment

This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFA1004102, in part by the National Natural Science Foundation of China under Grant 62250710682, and in part by the National Natural Science Foundation of China under Grant 62272210.

References

[1] B. H. Korte, J. Vygen, B. Korte, and J. Vygen, *Combinatorial Optimization*. Berlin, Germany: Springer, 2011.

[2] R. M. Karp, "Reducibility among combinatorial problems," in *Proc. Symp. Complexity Comput. Comput.*, 1972, pp. 85–103.

[3] J. Puchinger and G. R. Raidl, "Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification," in *Proc. Artif. Intell. Knowl. Eng. Appl.: Bioinspired Approach: 1st Int. Work-Conf. Interplay Between Natural Artif. Comput.*, 2005, pp. 41–53.

[4] G. Gutin and A. P. Punnen, *The Traveling Salesman Problem and Its Variations*. Berlin, Germany: Springer, 2006.

[5] K. Helsgaun, "An effective implementation of the lin–kernighan traveling salesman heuristic," *Eur. J. Oper. Res.*, vol. 126, no. 1, pp. 106–130, 2000.

[6] K. Helsgaun, "General K-opt submoves for the Lin-Kernighan TSP heuristic," *Math. Program. Comput.*, vol. 1, no. 2–3, pp. 119–163, 2009.

[7] Y. Nagata and S. Kobayashi, "A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem," *INFORMS J. Comput.*, vol. 25, no. 2, pp. 346–363, 2013.

[8] Y. Nagata, "Population diversity measures based on variable-order markov models for the traveling salesman problem," in *Proc. 14th Int. Conf. Parallel Problem Solving Nature*, 2016, pp. 973–983.

[9] É. D. Taillard and K. Helsgaun, "POPMUSIC for the travelling salesman problem," *Eur. J. Oper. Res.*, vol. 272, no. 2, pp. 420–429, 2019.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1106–1114.

[11] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. Int. Conf. Learn. Representations*, 2015.

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

[13] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: A methodological tour d'horizon," *Eur. J. Oper. Res.*, vol. 290, no. 2, pp. 405–421, 2021.

[14] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," in *Proc. Workshop Track Proc. ICLR*, 2017.

[15] Y. Kwon, J. Choo, I. Yoon, M. Park, D. Park, and Y. Gwon, "Matrix encoding networks for neural combinatorial optimization," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 5138–5149.

[16] L. Perron and V. Furnon, "OR-tools," Version v9.6, 2019. [Online]. Available: https://developers.google.com/optimization/

[17] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác, "Reinforcement learning for solving the vehicle routing problem," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 9839–9849.

[18] W. Kool, H. van Hoof, and M. Welling, "Attention, learn to solve routing problems!," in *Proc. Int. Conf. Learn. Representations*, 2019.

[19] X. Chen and Y. Tian, "Learning to perform local rewriting for combinatorial optimization," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 6281–6292.

[20] Y. Kwon, J. Choo, B. Kim, I. Yoon, Y. Gwon, and S. Min, "POMO: Policy optimization with multiple optima for reinforcement learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 21188–21198.

[21] Y. Ma et al., "Learning to iteratively solve routing problems with dual-aspect collaborative transformer," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 11096–11107.

[22] L. Accorsi, A. Lodi, and D. Vigo, "Guidelines for the computational testing of machine learning approaches to vehicle routing problems," *Oper. Res. Lett.*, vol. 50, no. 2, pp. 229–234, 2022.

[23] S. Liu, K. Tang, and X. Yao, "Automatic construction of parallel portfolios via explicit instance grouping," in *Proc. AAAI Conf. Artif. Intell.*, 2019, pp. 1560–1567.

[24] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Proc. Int. Conf. Learn. Intell. Optim.*, 2011, pp. 507–523.

[25] C. Ansótegui, M. Sellmann, and K. Tierney, "A gender-based genetic algorithm for the automatic configuration of algorithms," in *Proc. 15th Int. Conf . Princ. Pract. Constraint Program.*, 2009, pp. 142–157.

[26] M. López-Ibáñez, J. Dubois-Lacoste, L. P. Cáceres, M. Birattari, and T. Stützle, "The irace package: Iterated racing for automatic algorithm configuration," *Oper. Res. Perspectives*, vol. 3, pp. 43–58, 2016.

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

[28] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," in *Proc. Int. Conf. Learn. Representations*, 2017.

[29] 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. Integration Constraint Program., Artif. Intell., Oper. Res.*, 2018, pp. 170–181.

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

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

[32] 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, 1999.

[33] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev, "Reinforcement learning for combinatorial optimization: A survey," *Comput. Oper. Res.*, vol. 134, 2021, Art. no. 105400.

[34] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[35] B. Peng, J. Wang, and Z. Zhang, "A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems," in *Proc. 11th Int. Symp. Artif. Intell. Algorithms Appl.*, 2019, pp. 636–650.

[36] L. Xin, W. Song, Z. Cao, and J. Zhang, "Step-wise deep learning models for solving routing problems," *IEEE Trans. Ind. Informat.*, vol. 17, no. 7, pp. 4861–4871, Jul. 2021.

[37] 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.

[38] 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.

[39] 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.

[40] Q. Ma, S. Ge, D. He, D. Thaker, and I. Drori, "Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning," 2019, *arXiv:1911.04936*.

[41] 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.

[42] Y. Wu, W. Song, Z. Cao, J. Zhang, and A. Lim, "Learning improvement heuristics for solving routing problems," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 9, pp. 5057–5069, Sep. 2022.

[43] L. Xin, W. Song, Z. Cao, and J. Zhang, "Neurolkh: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 7472–7483.

[44] P. R. d. O. da Costa, J. Rhuggenaath, Y. Zhang, and A. Akcay, "Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning," in *Proc. Asian Conf. Mach. Learn.*, 2020, pp. 465–480.

[45] H. Lu, X. Zhang, and S. Yang, "A learning-based iterative method for solving vehicle routing problems," in *Proc. Int. Conf. Learn. Representations*, 2019.

[46] S. Salcedo-Sanz and X. Yao, "Assignment of cells to switches in a cellular mobile network using a hybrid hopfield network-genetic algorithm approach," *Appl. Soft Comput.*, vol. 8, no. 1, pp. 216–224, 2008.

[47] S. Salcedo-Sanz and X. Yao, "A hybrid hopfield network-genetic algorithm approach for the terminal assignment problem," *IEEE Trans. Syst. Man Cybern. Syst., Part B*, vol. 34, no. 6, pp. 2343–2353, Dec. 2004.

[48] A. Hottung and K. Tierney, "Neural large neighborhood search for the capacitated vehicle routing problem," in *Proc. ECAI*, 2019, pp. 443–450.

[49] K. Tang, Y. Mei, and X. Yao, "Memetic algorithm with extended neighborhood search for capacitated arc routing problems," *IEEE Trans. Evol.*, vol. 13, no. 5, pp. 1151–1166, Oct. 2009.

[50] X. Yao, "Simulated annealing with extended neighbourhood," *Int. J. Comput. Math.*, vol. 40, no. 3-4, pp. 169–189, 1991.

[51] P. Shaw, "A new local search algorithm providing high quality solutions to vehicle routing problems," *APES Group, Dept. Comput. Sci.*, vol. 46, 1997.

[52] M. Chen, L. Gao, Q. Chen, and Z. Liu, "Dynamic partial removal: A neural network heuristic for large neighborhood search," 2020, *arXiv:2005.09330*.

[53] L. Gao, M. Chen, Q. Chen, G. Luo, N. Zhu, and Z. Liu, "Learn to design the heuristics for vehicle routing problem," 2020, *arXiv:2002.08539*.

[54] J. Zheng, K. He, J. Zhou, Y. Jin, and C.-M. Li, "Combining reinforcement learning with lin-kernighan-helsgaun algorithm for the traveling salesman problem," in *Proc. AAAI Conf. Artif. Intell.*, 2021, pp. 12445–12452.

[55] J. Bossek, "netgen: Network generator for combinatorial graph problems," R package version, 2015.

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

[57] D. Applegate, R. Bixby, V. Chvatal, and W. Cook, "Concorde TSP solver," Version v03.12.19, 2006. [Online]. Available: https://www.math.uwaterloo.ca/tsp/concorde.html

[58] X. -F. Xie and J. Liu, "Multiagent optimization system for solving the traveling salesman problem (TSP)," *IEEE Trans. Syst. Man Cybern. Part B*, vol. 39, no. 2, pp. 489–502, Apr. 2009.

[59] S. Liu, K. Tang, and X. Yao, "Generative adversarial construction of parallel portfolios," *IEEE Trans. Cybern.*, vol. 52, no. 2, pp. 784–795, Feb. 2022.

[60] K. Tang, S. Liu, P. Yang, and X. Yao, "Few-shots parallel algorithm portfolio construction via co-evolution," *IEEE Trans. Evol. Comput.*, vol. 25, no. 3, pp. 595–607, Jun. 2021.

[61] C. H. Reilly, "Synthetic optimization problem generation: Show us the correlations!," *INFORMS J. Comput.*, vol. 21, no. 3, pp. 458–467, 2009.

[62] K. Smith-Miles and S. Bowly, "Generating new test instances by evolving in instance space," *Comput. Oper. Res.*, vol. 63, pp. 102–113, 2015.

[63] K. Tang, J. Wang, X. Li, and X. Yao, "A scalable approach to capacitated arc routing problems based on hierarchical decomposition," *IEEE Trans. Cybern.*, vol. 47, no. 11, pp. 3928–3940, Nov. 2017.