

Few-Shots Parallel Algorithm Portfolio Construction via Co-Evolution

Ke Tang¹, Senior Member, IEEE, Shengcai Liu², Member, IEEE,
Peng Yang³, Member, IEEE, and Xin Yao⁴, Fellow, IEEE

Abstract—Generalization, i.e., the ability of solving problem instances that are not available during the system design and development phase, is a critical goal for intelligent systems. A typical way to achieve good generalization is to learn a model from vast data. In the context of heuristic search, such a paradigm could be implemented as configuring the parameters of a parallel algorithm portfolio (PAP) based on a set of “training” problem instances, which is often referred to as PAP construction. However, compared to the traditional machine learning, PAP construction often suffers from the lack of training instances, and the obtained PAPs may fail to generalize well. This article proposes a novel competitive co-evolution scheme, named co-evolution of parameterized search (CEPS), as a remedy to this challenge. By co-evolving a configuration population and an instance population, CEPS is capable of obtaining generalizable PAPs with few training instances. The advantage of CEPS in improving generalization is analytically shown in this article. Two concrete algorithms, namely, CEPS-TSP and CEPS-VRPSPDTW, are presented for the traveling salesman problem (TSP) and the vehicle routing problem with simultaneous pickup-delivery and time windows (VRPSPDTW), respectively. The experimental results show that CEPS has led to better generalization, and even managed to find new best-known solutions for some instances.

Index Terms—Algorithm configuration, automatic parameter tuning, co-evolution, parallel algorithm portfolios (PAPs), vehicle routing problems.

Manuscript received July 1, 2020; revised October 20, 2020, December 30, 2020, and February 3, 2021; accepted February 5, 2021. Date of publication February 16, 2021; date of current version May 28, 2021. This work was supported in part by the Guangdong Provincial Key Laboratory under Grant 2020B121201001; in part by the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant 2017ZT07X386; in part by the Shenzhen Peacock Plan under Grant KQTD2016112514355531; in part by the Science and Technology Commission of Shanghai Municipality under Grant 19511120600; in part by the National Leading Youth Talent Support Program of China; and in part by the MOE University Scientific-Technological Innovation Plan Program. (Corresponding author: Ke Tang.)

Ke Tang and Shengcai Liu are with the Guangdong Key Laboratory of Brain-Inspired Intelligent Computation, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China, and also with the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: tangk3@sustech.edu.cn; liusc3@sustech.edu.cn).

Peng Yang and Xin Yao are with the Guangdong Key Laboratory of Brain-Inspired Intelligent Computation, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: yangp@sustech.edu.cn; xiny@sustech.edu.cn).

Digital Object Identifier 10.1109/TEVC.2021.3059661

I. INTRODUCTION

IN THE past decades, search methods have become major approaches for tackling various computationally hard problems. Most, if not all, established search methods, from specialized heuristic algorithms tailored for a particular problem class, e.g., the Lin-Kernighan (LK) heuristic for the traveling salesman problem (TSP) [1], to general algorithmic frameworks, e.g., evolutionary algorithms, share a common feature. That is, they are parameterized algorithms, which means they involve parameters that need to be configured by users before the algorithm is applied to a problem.

Although theoretical analyzes for many parameterized algorithms have offered worst or average bounds on their performance, their actual performance, in practice, is in many cases highly sensitive to the settings of parameters [2]–[5]. More importantly, finding the optimal configuration, i.e., parameter setting, requires knowledge of both the algorithm and the problem to solve, which cannot be done manually with ease. Hence, a lot of efforts have been made to automate this procedure, often dubbed automatic parameter tuning [5], [6] when the algorithms have relatively few parameters with mostly real-valued domains, or automatic algorithm configuration [3], [7]–[10] when the algorithms have more types (e.g., ordinal and categorical) of parameters. These methods essentially involve a high-level iterative generate-and-test process. To be specific, given a set of instances from the target problem class, different configurations are iteratively generated and tested on the instance set. Upon termination, the process outputs the configuration that performs the best on the instance set. Since a configuration fully instantiates a parameterized algorithm, for brevity, henceforth, we will use the term “configuration” to directly denote the resultant solver specified by it.

Built upon automatic algorithm configuration, the automatic construction of parallel algorithm portfolios (PAPs) [11]–[15] seeks to identify a set of configurations to form a PAP. Each configuration in the PAP is called a component solver. To solve a problem instance, all the component solvers are run independently, typically in parallel, to get multiple solutions. Then, the best solution will be taken as the output of PAP. Although a PAP would consume much more computational resources than a single-configuration solver, it has two important advantages. First, the performance of a PAP on any given instance is the best performance achieved among its component solvers on the instance. In other words, by exploiting the complementarity between the component solvers, a PAP could

achieve a much stronger overall performance than any of its component solver. Second, considering the great development of parallel computing architectures [16] (e.g., multicore CPUs) over the last decade, exploiting parallelism has become very important in designing efficient solvers for computationally hard problems. PAPs employ parallel solution strategies, and thus allow exploiting modern high-performance computing facilities in an extremely simple way.

From the practical point of view, a PAP construction method is expected to identify a PAP that generalizes well, i.e., performs well not only on the instance set used during the tuning phase but also on unseen instances of the same problem class. The reason is that intelligent systems, which incorporate parameterized search algorithms as a module, are seldom built to address a few specific problem instances, but for a whole target problem class, and it is unlikely to know in advance the exact problem instances that a system will encounter in practice. The need for generalization requires the instance set used for PAP construction to be sufficiently large such that it consists of good representatives of all instances of the target problem class. Unfortunately, in a real-world scenario, PAP construction is very likely to face the few-shots challenge. That is, the available instance set is not only of small size but also may not well represent the target problem class. For example, the widely studied TSP benchmark suites (i.e., TSPLib [17]) consist of a few hundred TSP instances, while there could be millions of possibilities for concrete TSP instances even if only considering a fixed number of cities. In consequence, the more powerful of a PAP construction method, the higher risk that the obtained PAP will overfit the instances involved in the tuning process.

This article suggests that the pursuit of generalizable PAPs could be modeled as a co-evolutionary system, in which two internal populations, representing the configurations (the PAP) and the problem instances, respectively, compete with each other during the evolution course. The evolution of the latter promotes exploration in the instance space of the target problem class to generate synthetic instances that exploit the weakness of the former. The former, on the other hand, improves itself by identifying configurations that could better handle the latter. In this way, the configuration population (the PAP) is encouraged to evolve toward achieving good performance on as many instances of the target problem class as possible, i.e., toward better generalization. Specifically, the contributions of this article include the following.

- 1) A novel PAP construction framework, namely, co-evolution of parameterized search (CEPS), is proposed. It is also shown that CEPS approximates a process that minimizes the upper bound, i.e., a tractable surrogate, of the generalization performance.
- 2) To demonstrate the implementation details of CEPS as well as to assess its potential, concrete instantiations are also presented for two hard optimization problems, i.e., TSP and the vehicle routing problem with simultaneous pickup–delivery and time windows (VRPSPDTWs) [18]. Computational studies confirm that CEPS is able to obtain PAPs with better generalization performance.

- 3) The proposal of CEPS extends the realm of co-evolution, for the first time, to evolving algorithm configurations and problem instances. Since CEPS does not invoke domain-specific knowledge, its potential applications can go beyond optimization problems, even to planning and learning problems.

The remainder of this article is organized as follows. Section II introduces the challenge of seeking generalizable PAPs, existing PAP construction methods, as well as the theoretical insight behind CEPS. Section III presents the CEPS framework. Section IV presents its instantiations for TSP and VRPSPDTW. Computational studies on these two problems are presented in Section V. Threats to validity of this study are discussed in Section VI. Section VII concludes the article with discussions.

II. PARAMETERIZED SOLVERS MADE GENERALIZABLE

A. Notations and Problem Description

Assume a PAP is to be built for a problem class (e.g., TSP), for which an instance of the problem class is denoted as s , and the set of all possible s is denoted as Ω . Given a parameterized algorithm, each component solver of the PAP is a configuration (full instantiation) of the algorithm. Generally speaking, the parameterized algorithm can be any concrete computational process, e.g., a traditional heuristic search process, such as the LK Heuristic for TSP or even a neural network [19]–[21] that outputs a solution for a given instance of the target problem class. Let θ denote a configuration and let Θ denote a PAP that contains K different configurations (component solvers), i.e., $\Theta = \{\theta_1, \dots, \theta_K\}$. The quality of a configuration θ on a given instance s is denoted as $f(s, \theta)$, which is a performance indicator of the corresponding solver on the instance. This indicator could concern many aspects, e.g., the quality of the obtained solution [10], the CPU time required to achieve a solution above a given quality threshold [7], or even be stated in a multiobjective form [22]. The performance of a PAP Θ on an instance s , denoted as $f(s, \Theta)$, is the best performance achieved among its component solvers $\theta_1, \dots, \theta_k$ on s [assuming the smaller $f(s, \theta)$, the better]

$$f(s, \Theta) := \min\{f(s, \theta_1), \dots, f(s, \theta_K)\}. \quad (1)$$

Following the above definitions, optimizing the generalization performance of a PAP can be stated as:

$$\min_{\Theta} J(\Theta) := \int_{s \in \Omega} f(s, \Theta) p(s) ds \quad (2)$$

where $p(s)$ stands for the prior distribution of s . Since, in practice, the prior distribution is usually unknown, a uniform distribution can be assumed without loss of generality. Equation (2) can be then simplified to (3) by omitting a normalization constant

$$\min_{\Theta} J(\Theta) := \int_{s \in \Omega} f(s, \Theta) ds. \quad (3)$$

The challenge with (2) and (3) is that in practice they cannot be directly optimized since the set Ω is generally unavailable. Instead, only a set of “training” instances, i.e., a subset $T \subset \Omega$,

is given for the purpose of constructing Θ . In fact, the so-called overtuning phenomenon [23], [24], which is analogous to the overfitting phenomenon in machine learning, has been observed when the size of the training instance set is rather limited (i.e., few-shots challenge). That is, the test (generalization) performance of the obtained configurations is arbitrarily bad even if their performance on the training set is excellent. Even worse, given a T collected from real world, it is nontrivial to know how to verify whether it is a good representative of Ω . In case the training instance set is too small, or is not a good representative of the whole problem class, the best PAP obtained with it would fail to generalize.

B. Related Work

Currently, there exist several approaches for PAP construction, namely, GLOBAL [25], PARHYDRA [25], [26], CLUSTERING [27], and PCIT [13]. GLOBAL considers PAP construction as an algorithm configuration problem by treating Θ as a parameterized algorithm. By this means, existing automatic algorithm configuration tools could be directly utilized to configure all the component solvers of Θ simultaneously. In comparison, PARHYDRA constructs Θ iteratively by identifying a single component solver in each iteration that maximizes marginal performance contribution to the current PAP. CLUSTERING and PCIT are two approaches based on instance grouping. That is, they both first split the training set into disjoint subsets, and then identify a component solver on each subset. The former splits the training set by clustering training instances based on their feature-vector representations, while the latter splits the training set uniform randomly and will adjust the instance grouping by transferring instances between subsets during the PAP construction process.

Other than PAP, another important way of utilizing an algorithm portfolio to achieve stronger overall performance is algorithm selection (AS), which seeks to select, from a given algorithm portfolio, the best suited solver to solve an instance. Specifically, the algorithm selector is usually built by training machine learning models based on the feature sets of training instances. Over the last decades, AS has been successfully applied to many computationally hard problems, such as Boolean satisfiability problems [28] and TSP [29], [30]. A comprehensive survey on AS can be found in [31].

To the best of our knowledge, the few-shots challenge has not been investigated yet in the literature. That is, all the methods mentioned above assume that the training instances could sufficiently represent that target problem class. However, as aforementioned, such an assumption could not be always true since in some cases, we might only have scarce or biased training instances.

C. Enhancing Generalization With Synthetic Instances

A natural idea to tackle the few-shots challenge is to augment T with a set of synthetic instances, say T' , such that the PAP obtained with $T \cup T'$ would generalize better than that obtained with T . This idea is generally valid because if the size of T' continues to grow, $T \cup T'$ will eventually approach Ω .

Hence, the key question is how a generalizable PAP could be obtained with a sufficiently small T' . This question can be restated as: how to generate synthetic training instances, such that the generalization of the obtained PAP could be improved as much as possible with a T' of (say predefined) small size.

Given a parameterized algorithm, suppose a PAP Θ has been obtained as the best-performing PAP on T . A synthetic instance set T' is to be generated, with the aim that a new PAP Θ' obtained with $T \cup T'$ would outperform Θ in terms of generalization as much as possible. Ignoring the inner optimization/tuning process with which Θ' and Θ are obtained, generating high quality T' could be more formally stated as another optimization problem in (4) as follows:

$$\begin{aligned} \min_{T'} \{J(\Theta') - J(\Theta)\} &:= \int_{s \in \Omega} f(s, \Theta') ds - \int_{s \in \Omega} f(s, \Theta) ds \\ &= \left[\sum_{s \in T} f(s, \Theta') + \sum_{s \in T'} f(s, \Theta') + \int_{s \in \Omega \setminus (T \cup T')} f(s, \Theta') ds \right] \\ &\quad - \left[\sum_{s \in T} f(s, \Theta) + \sum_{s \in T'} f(s, \Theta) + \int_{s \in \Omega \setminus (T \cup T')} f(s, \Theta) ds \right]. \end{aligned} \quad (4)$$

Equation (4) aims at achieving the largest improvement over $J(\Theta)$, which is a constant since Θ has been obtained with T . Since Θ' is obtained with $T \cup T'$, we further assume that for any $s \in \Omega$, $f(s, \Theta') \leq f(s, \Theta)$. Although this is a rather restrictive assumption, it will be shown later that it could be fulfilled when Θ is a subset of Θ' . Applying this assumption to the right-hand side of (4), we have

$$\begin{aligned} \sum_{s \in T} [f(s, \Theta') - f(s, \Theta)] &\leq 0 \\ \sum_{s \in T'} [f(s, \Theta') - f(s, \Theta)] &\leq 0 \\ \int_{s \in \Omega \setminus (T \cup T')} [f(s, \Theta') - f(s, \Theta)] ds &\leq 0. \end{aligned} \quad (5)$$

Considering that in the right-hand side of (4), $\Omega \setminus (T \cup T')$ is unknown, we thus discard the terms regarding $\Omega \setminus (T \cup T')$ and retain the ones regarding T and T' . By inequality (5), the discarded terms are nonpositive, we then have

$$J(\Theta') - J(\Theta) \leq \sum_{s \in T \cup T'} [f(s, \Theta') - f(s, \Theta)]. \quad (6)$$

Inequality (6) gives an upper bound of $J(\Theta') - J(\Theta)$ that depends on the current instance set T , target instance set T' , current PAP Θ , and new PAP Θ' . Note the upper bound is always nonpositive by inequality (5), which means if the assumption holds, the new PAP Θ' is guaranteed to generalize better than the current PAP Θ . More importantly, considering that neither $J(\Theta')$ nor $J(\Theta)$ can be precisely measured in practice, the upper bound in Inequality (6) provides a measurable surrogate for minimizing $J(\Theta') - J(\Theta)$, such that even larger performance improvement could be achieved than only relying on the assumption. Therefore, given a training instance set T and a PAP Θ obtained with T , an improved PAP Θ' (in terms of generalization performance) could be obtained

with a strategy with two steps to minimize the upper bound in inequality (6).

- 1) Identify T' that maximizes $\sum_{s \in T \cup T'} f(s, \Theta)$ [this is equivalent to maximizing $\sum_{s \in T'} f(s, \Theta)$ since $\sum_{s \in T} f(s, \Theta)$ is a constant given that T and Θ are fixed].
- 2) Identify Θ' that minimizes $\sum_{s \in T \cup T'} f(s, \Theta')$ [note that once T' is generated, the term $\sum_{s \in T \cup T'} f(s, \Theta)$ is a constant and can be omitted].

The above two steps naturally serve as the core building block of an iterative process that gradually seek PAPs with better generalization performance. There could be many ways to design such an iterative process. Among them competitive co-evolution [32] provides a readily available framework. That is, one can maintain an instance population (representing the instance set) and a configuration population (representing the PAP). In each iteration, the two populations alternately evolve and compete with each other, i.e., the instance population evolves to identify T' and the configuration population evolves to identify Θ' .

Recall that the two-step improvement strategy is derived from the assumption that $f(s, \Theta') \leq f(s, \Theta)$ for any $s \in \Omega$. This assumption holds if Θ is a subset of Θ' because by definition of PAP (1), we have: $f(s, \Theta') = \min\{f(s, \Theta), \min_{\theta \in \Theta' \setminus \Theta} f(s, \theta)\} \leq f(s, \Theta)$. Following this, one could further design the evolution of the PAP (the configuration population) as identifying new configurations to insert into the current PAP Θ , such that the new PAP Θ' , which is a superset of Θ , minimizes $\sum_{s \in T \cup T'} f(s, \Theta')$. However, in practice such a mechanism could suffer from the PAP-size issue. That is, the number of the component solvers in the PAP will keep increasing as the co-evolution proceeds. Recall that a PAP runs its component solvers in parallel; thus, its size is mandatorily limited by the available computational resources (e.g., the number of available CPU cores) and thus cannot grow infinitely. A natural way to avoid this issue is to first remove some configurations from PAP Θ , resulting in a temporary PAP $\bar{\Theta}$, and then identify new configurations to insert into $\bar{\Theta}$, such that the final PAP Θ' is of the same size as Θ . However, this approach no longer guarantees the validity of the above assumption. As a consequence, Θ' may generalize worse than Θ . A remedy to prevent this as much as possible is to increase redundancy in the evolution of PAP. More specifically, one could repeat the configuration-removal procedure to Θ for n times, leading to n temporary PAPs, $\bar{\Theta}_1, \dots, \bar{\Theta}_n$; then for each temporary PAP $\bar{\Theta}_i$, the new configurations are identified and inserted, leading to n new PAPs, $\Theta'_1, \dots, \Theta'_n$, each of which is of the same size as Θ ; finally, the PAP among them that performs best against $T \cup T'$ is retained.

III. CO-EVOLUTION OF PARAMETERIZED SEARCH

By incorporating the above-described procedure into the co-evolution process, we arrive at the proposed CEPS framework, as demonstrated in Algorithm 1. In general, CEPS consists of two major phases, i.e., an initialization phase (lines 2–7), and a co-evolution phase (lines 8–27), which could be further subdivided into alternating between the evolution of

Algorithm 1: General Framework of CEPS

```

input: training set  $T$ ; number of component solvers,  $K$ ;
         number of temporary PAPs,  $n$ ; maximum number of
         iterations,  $MaxIte$ 
output: the final configuration population (PAP)  $\Theta$ 
1 /* -----Initialization----- */
2 Randomly sample a set  $C$  from the configuration space, and
   test all the selected configurations on  $T$ ;
3  $\Theta \leftarrow \emptyset$ ;
4 for  $i \leftarrow 1$  to  $K$  do
5   Find  $\theta_i$  from  $C$ , with the target minimizing
      $\frac{1}{|T|} \sum_{s \in T} f(s, \Theta \cup \{\theta_i\})$ ;
6    $\Theta \leftarrow \Theta \cup \{\theta_i\}$ ;
7 end
8 for  $ite \leftarrow 1$  to  $MaxIte$  do
9   /* -----Evolution of  $\Theta$ ----- */
10  for  $i \leftarrow 1$  to  $n$  do
11    Randomly select  $\theta \in \Theta$ , and let  $\bar{\Theta}_i \leftarrow \Theta \setminus \{\theta\}$ ;
12    Use SMAC to identify  $\theta'$ , with the target minimizing
       $\frac{1}{|T|} \sum_{s \in T} f(s, \bar{\Theta}_i \cup \{\theta'\})$ ;
13     $\Theta'_i \leftarrow \bar{\Theta}_i \cup \{\theta'\}$ ;
14  end
15   $\Theta \leftarrow$  the best-performing PAP among  $\Theta'_1, \dots, \Theta'_n$ ;
16  /* -----Evolution of  $T$ ----- */
17  if  $ite = MaxIte$  then break;
18   $T' \leftarrow$  create a copy of  $T$ ;
19  Assign the fitness of each  $s \in T'$  as  $f(s, \Theta)$ ;
20  while not terminated do
21     $s' \leftarrow$  randomly select  $s \in T'$ , and mutate  $s$ ;
22    Test  $s'$  with  $\Theta$  and assign the fitness of  $s'$  as  $f(s', \Theta)$ ;
23     $s^* \leftarrow$  randomly select one from all the instances in  $T'$ 
      with lower fitness than  $s'$ ;
24     $T \leftarrow T \setminus \{s^*\} \cup \{s'\}$ ;
25  end
26   $T \leftarrow T' \cup T$ ;
27 end
28 return  $\Theta$ 

```

the configuration population (representing the PAP) (lines 10–15) and the evolution of the instance population (representing the training instances) (lines 17–26) for $MaxIte$ iterations in total. These modules are detailed as follows.

1) *Initialization:* Given an initial training instance set T , a simple greedy strategy is adopted to initialize a configuration population (PAP) Θ of size K . First, a set of candidate configurations C is randomly sampled from the configuration space and tested on the training set T (line 2). Then, starting from an empty set (line 3), Θ is built iteratively (lines 4–7). At each iteration, the configuration whose inclusion into Θ leads to the largest performance improvement is selected from C (line 5) and inserted into Θ (line 6). The process terminates when K configurations have been selected.

2) *Evolution of the Configuration Population:* Given a configuration population Θ , n temporary PAPs, $\bar{\Theta}_1, \dots, \bar{\Theta}_n$, are first generated by repeatedly randomly removing a configuration from Θ (line 11). Then, for each $\bar{\Theta}_i$, an existing automatic algorithm configuration tool, namely, SMAC [3], is used to search in the configuration space to find a new configuration θ' with the target that the inclusion of θ' into $\bar{\Theta}_i$ leads to the minimization of the performance of the resultant PAP Θ'_i on the training set (lines 12 and 13). Finally, the

Algorithm 2: Instance Mutation Operator in CEPS-TSP

```

input: instance  $s$ 
output: mutated instance  $s$ 
1 Let  $N$  be the number of cities in  $s$ , which is then represented
  by  $\{(x_1, y_1), \dots, (x_N, y_N)\}$ ;
2  $x_{\min} \leftarrow \min\{x_1, \dots, x_N\}$ ;  $x_{\max} \leftarrow \max\{x_1, \dots, x_N\}$ ;
3  $y_{\min} \leftarrow \min\{y_1, \dots, y_N\}$ ;  $y_{\max} \leftarrow \max\{y_1, \dots, y_N\}$ ;
4 for  $i \leftarrow 1$  to  $N$  do
5   Generate a random number  $r \in [0, 1]$ ;
6   if  $r \leq 0.9$  then
7     Sample  $\Delta \sim \mathcal{N}(0, [0.025 \cdot (x_{\max} - x_{\min})]^2)$ ;
8      $x_i \leftarrow x_i + \Delta$ ;
9     Sample  $\Delta \sim \mathcal{N}(0, [0.025 \cdot (y_{\max} - y_{\min})]^2)$ ;
10     $y_i \leftarrow y_i + \Delta$ ;
11  else
12    Sample  $x' \sim \mathcal{U}(x_{\min}, x_{\max})$ ;
13     $x_i \leftarrow x'$ ;
14    Sample  $y' \sim \mathcal{U}(y_{\min}, y_{\max})$ ;
15     $y_i \leftarrow y'$ ;
16  end
17 end
18 return  $s$ 

```

best-performing PAP among the n new PAPs $\Theta'_1, \dots, \Theta'_n$ will replace Θ (line 15). From the perspective of evolutionary computation, the above procedure could be seemed as mutation to Θ , with SMAC employed as the mutation operator.

3) *Evolution of the Instance Population:* In this phase, CEPS first creates a copy of T , i.e., T' , which will serve as the initial instance population hereafter (line 18). Since the aim of the evolution of the instance population is to identify T' that are hard for Θ , i.e., maximizing $\sum_{s \in T'} f(s, \Theta)$, each instance in T' is assigned with a fitness as the performance of Θ on it (line 19)—the worse the performance, the higher the fitness. In each generation of the evolution of the instance population, CEPS first randomly selects an instance s from T' as the parent and mutates it to generate an offspring s' (line 21), which is then evaluated against the configuration population (line 22). Finally, CEPS uses s' to randomly replace an instance in T' that has lower fitness than s' (lines 23 and 24). In this way, as the number of generations increases, the average fitness of instances in T' will gradually increase, meaning that the instances in T' will be harder and harder for the configuration population. When the evolution of the instance population ends, the final T' will be merged into the training set (line 26), which will be used for obtaining Θ' in the next iteration of the co-evolution. Note in the last iteration (i.e., the *MaxIteth* iteration) of the co-evolution phase, evolution of the instance population is skipped (line 17) because there is no need to generate more instances since the final configuration population has been constructed completely.

IV. INSTANTIATIONS FOR TSP AND VRPSPDTW

Algorithm 1 is a rather generic framework since the representations of both populations depend on the target parameterized algorithm and the target problem class, respectively. The mutation operator for the instance population as well as the fitness function also depend on target problem class. In

this article, two instantiations of CEPS, namely, CEPS-TSP and CEPS-VRPSPDTW, have been developed for TSP and VRPSPDTW problems, respectively. These two target problem classes are chosen because, as a classic NP-hard problem, TSP is one of the most widely investigated benchmarking problems in academia. In comparison, VRPSPDTW is a much more complex routing problem that takes real-world requirements into account. The significant difference between these two problems could provide a good context for assessing CEPS.

A. CEPS-TSP

Given a list of cities and the distances between each pair of cities, the target of TSP is to find the shortest route that visits each city and returns to the origin city. Specifically, the symmetric TSP with distances in a 2-D Euclidean space is considered here.

1) *Instance Mutation Operator:* Each of such TSP instance is represented by a list of (x, y) coordinates with each coordinate as a city. An operator widely used for generating TSP instances (see [33]) is employed as the instance mutation operator of CEPS-TSP. As illustrated in Algorithm 2, the mutation operator works as follows. Let x_{\min} and x_{\max} , y_{\min} and y_{\max} , be the minimum and the maximum of the “ x ” values and the “ y ” values across all coordinates of a given instance s , respectively. When applying mutation to s , for each coordinate (x, y) in s , x and y are offset with probability 0.9 by the step sizes sampled from $\mathcal{N}(0, [0.025 \cdot (x_{\max} - x_{\min})]^2)$ and $\mathcal{N}(0, [0.025 \cdot (y_{\max} - y_{\min})]^2)$, respectively, and with probability 0.1, x and y are replaced by new values sampled from $\mathcal{U}(x_{\min}, x_{\max})$ and $\mathcal{U}(y_{\min}, y_{\max})$, respectively. $\mathcal{N}(\mu, \sigma^2)$ refers to normal distribution with mean μ and variance σ^2 , and $\mathcal{U}(a, b)$ refers to normal distribution defined on closed interval $[a, b]$.

2) *Parameterized Algorithm:* The adopted parameterized algorithm is the Helsgaun’s LK Heuristic (LKH) [34] version 2.0.7 (with 23 parameters), one of the state-of-the-art inexact solver for such TSP.

3) *Fitness Function:* For TSP, the penalized average runtime with penalty factor 10 (PAR-10) [7] is considered as the performance indicator. The smaller the PAR-10, the better. More specifically, the performance of a configuration θ on an instance s , i.e., $f(s, \theta)$, is the penalized runtime needed by θ to solve s . In particular, when running θ on s , the run would be terminated as soon as the optimal solution of s is found or after a cutoff time of 10 s. In the first case, the run is considered successful and $f(s, \theta)$ is exactly the recorded runtime; in the second case, the run is considered timeout and $f(s, \theta)$ is the cutoff time multiplied by the penalty factor 10, i.e., $10 \times 10 = 100$ s. Based on $f(s, \theta)$, the performance of a PAP solver Θ on an instance s , i.e., $f(s, \Theta)$, as defined in (1), is $\min\{f(s, \theta) | \theta \in \Theta\}$, which is the fitness function used in the evolution of the instance population in CEPS-TSP (lines 19 and 22 in Algorithm 1). Finally, the performance of a solver (a single configuration or a PAP solver) on an instance set is the average of the penalized runtime over all instances in the set, which is directly used for fitness evaluation in CEPS-TSP to compare PAPs constructed with the configuration population (line 15 of Algorithm 1).

B. CEPS-VRPSPDTW

Given a number of customers who require both pickup service and delivery service within a certain time window, the target of VRPSPDTW [18] is to send out a fleet of capacitated vehicles, which are stationed at a depot, to meet the customer demands with the minimum total cost. More specifically, VRPSPDTW is defined on a complete graph $G = (V, E)$ with $V = \{0, 1, 2, \dots, N\}$ as the node set and E as the arc set defined between each pair of nodes, i.e., $E = \{(i, j) | i, j \in V, i \neq j\}$. For convenience, the depot is denoted as 0 and the customers are denoted as $1, \dots, N$. Each node $i \in V$ has a coordinate (x_i, y_i) and the distance between i and j , denoted as $c_{i,j}$, is the Euclidean distance. In addition to the coordinate, each customer is associated with five attributes, i.e., a delivery demand d_i , a pickup demand p_i , a time window $[a_i, b_i]$, and a service time s_i . d_i represents the amount of goods to deliver from the depot to customer i and p_i represents the amount of goods to pick up from customer i to be delivered to the depot. a_i and b_i define the start and the end of the time window in which the customer receives service. The time windows are treated as hard constraints. That is, arrival of a vehicle at the customer i before a_i results in a wait before service can begin; while arrival after b_i is infeasible. Finally, s_i is the time spent by the vehicle to load/unload goods at customer i . A fleet of J identical vehicles, each with a capacity of Q and dispatching cost c_d , is initially located at the depot. Each vehicle starts at the depot and then serve the customers, and finally returns to the depot. For convenience, the depot 0 is also associated with five attributes, in which a_0 and b_0 are the earliest time the vehicles can depart from the depot and the latest time the vehicles can return to the depot, respectively, and $d_0 = p_0 = s_0 = 0$.

A solution S to VRPSPDTW could be represented by a set of vehicle routes, i.e., $S = \{R_1, R_2, \dots, R_K\}$, in which each route R_i consists of a sequence of nodes that the vehicle visits, i.e., $R_i = (h_{i,1}, h_{i,2}, \dots, h_{i,L_i})$, where $h_{i,j}$ is the j th node visited in R_i , and L_i is the length of R_i . Let $TD(R_i)$ denote the total travel distance in R_i , and let $\text{load}(R_i)$ denote the highest load on the vehicle that occurs in R_i . Let $\text{arr}(h_{i,j})$ and $\text{dep}(h_{i,j})$ denote the time of arrival at $h_{i,j}$ and the time of departure from $h_{i,j}$, respectively.

The total cost of S consists of two parts: 1) the dispatching cost of the used vehicles, which is $K \cdot c_d$ and 2) the transportation cost, which is the total travel distance in S multiplied by unit transportation cost u . The objective of the VRPSPDTW problem is to find routes for vehicles that serve all the customers at a minimal cost, which is presented in (7) as follows:

$$\begin{aligned} \min_S \quad & TC(S) := \sum_{i=1}^K [c_d + TD(R_i) \cdot u] \\ \text{s.t.:} \quad & K \leq J \\ & h_{i,1} = h_{i,L(i)} = 0, 1 \leq i \leq K \\ & \sum_{i=1}^K \sum_{j=2}^{L_i-1} I[h_{i,j} = e] = 1, 1 \leq e \leq N \\ & \text{load}(R_i) \leq Q, 1 \leq i \leq K \end{aligned}$$

$$\begin{aligned} \text{dep}(h_{i,1}) &\geq a_0, 1 \leq i \leq K \\ \text{arr}(h_{i,j}) &\leq b_{h_{i,j}}, 1 \leq i \leq K, 2 \leq j \leq L_i \end{aligned} \quad (7)$$

where the constraints are: 1) the number of used vehicles must be smaller than the number of available ones; 2) each customer must be served exactly once; 3) the vehicle cannot be overloaded during transportation; 4) the vehicles can only serve after the start of the time window of the depot, and must return to the depot before the end of the time window of the depot; and 5) the service of the vehicle to each customer must be performed within that customer's time window.

We consider a practical application scenario from the JD logistics company. Consider a VRPSPDTW solver that needs to solve a VRPSPDTW instance every day. The company has about 3000 customers in total in the city, but only about 13% of its customers (i.e., 400) require service per day. Therefore, for the solver, the different VRPSPDTW instances it faces have the following connections: 1) the location and the time window of the depot are unchanged, and the vehicle fleet is unchanged; 2) the locations of the customers will change; 3) the time windows of the customers will change; and 4) the delivery and pickup demands of the customers will change.

1) *Instance Mutation Operator*: Based on the above observation, we design a specialized mutation operator for VRPSPDTW, as presented in Algorithm 3. First, the coordinate mutation used in CEPS-TSP is also used here. Moreover, for the pickup demand p_i and the delivery demand d_i of each customer, they are replaced by new values sampled from $\mathcal{U}(p_{\min}, p_{\max})$ and $\mathcal{U}(d_{\min}, d_{\max})$, respectively, where p_{\min} and p_{\max} , u_{\min} and u_{\max} , are the minimum and the maximum of the “ p ” value and the “ d ” values across all customers of s , respectively. For the time window $[a_i, b_i]$ of each customer, a_i and b_i are offset by the step sizes sampled from $\mathcal{N}(0, (0.025 \cdot (b_0 - a_0)^2))$, where a_0 and b_0 are the earliest time that the vehicles can depart from the depot and the latest time that the vehicles can return to the depot.

2) *Parameterized Algorithm*: The adopted parameterized algorithm for VRPSPDTW is a powerful co-evolutionary genetic algorithm (Co-GA) proposed by [18] (with 12 parameters).

3) *Fitness Function*: For VRPSPDTW, the penalized average normalized cost (PANC), is considered as the performance indicator. The smaller the PANC, the better. More specifically, the performance of a configuration θ on an instance s , i.e., $f(s, \theta)$, is the penalized normalized cost of the solution found by θ . In particular, the run of θ on s would be terminated after a cutoff time of 150 s. Assume θ successfully finds a feasible solution of cost c to s . Considering for different VRPSPDTW instances, the scales of the solution costs may vary significantly, thus the “normalized cost” is introduced to replace c

$$f(s, \theta) = \frac{c}{\text{mean_distance}(s)} \quad (8)$$

where $\text{mean_distance}(s)$ is the average distance between all pairs of customers in instance s . In case that θ fails to find a feasible solution to s within the cutoff time, the corresponding run is considered timeout and $f(s, \theta)$ will be set to a large

Algorithm 3: Instance Mutation Operator in CEPS-VRPSPDTW

```

input: instance  $s$ 
output: mutated instance  $s$ 
1 Let  $N$  be the number of customers, which are represented
  by  $\{(x_0, y_0, d_0, p_0, a_0, b_0, s_0), \dots, (x_N, y_N, d_N, p_N, a_N, b_N, s_N)\}$ ;
2  $x_{min} \leftarrow \min\{x_1, \dots, x_N\}$ ;  $x_{max} \leftarrow \max\{x_1, \dots, x_N\}$ ;
3  $y_{min} \leftarrow \min\{y_1, \dots, y_N\}$ ;  $y_{max} \leftarrow \max\{y_1, \dots, y_N\}$ ;
4  $p_{min} \leftarrow \min\{p_1, \dots, p_N\}$ ;  $p_{max} \leftarrow \max\{p_1, \dots, p_N\}$ ;
5  $u_{min} \leftarrow \min\{u_1, \dots, u_N\}$ ;  $u_{max} \leftarrow \max\{u_1, \dots, u_N\}$ ;
6 for  $i \leftarrow 1$  to  $N$  do
7   /* ---Coordinate Mutation--- */
8   Generate a random number  $r \in [0, 1]$ ;
9   if  $r \leq 0.9$  then
10    Sample  $\Delta \sim \mathcal{N}(0, [0.025 \cdot (x_{max} - x_{min})]^2)$ ;
11     $x_i \leftarrow x_i + \Delta$ ;
12    Sample  $\Delta \sim \mathcal{N}(0, [0.025 \cdot (y_{max} - y_{min})]^2)$ ;
13     $y_i \leftarrow y_i + \Delta$ ;
14  else
15    Sample  $x' \sim \mathcal{U}(x_{min}, x_{max})$ ;
16     $x_i \leftarrow x'$ ;
17    Sample  $y' \sim \mathcal{U}(y_{min}, y_{max})$ ;
18     $y_i \leftarrow y'$ ;
19  end
20  /* ---Demand Mutation----- */
21  Sample  $p' \sim \mathcal{U}(p_{min}, p_{max})$ ;
22   $p_i \leftarrow p'$ ;
23  Sample  $d' \sim \mathcal{U}(d_{min}, d_{max})$ ;
24   $d_i \leftarrow d'$ ;
25  /* ---Time-window Mutation--- */
26  Sample  $\Delta_1, \Delta_2 \sim \mathcal{N}(0, (0.025 \cdot (b_0 - a_0))^2)$ ;
27   $a_i \leftarrow a_i + \Delta_1$ ;
28   $b_i \leftarrow b_i + \Delta_2$ ;
29 end
30 return  $s$ 

```

penalty value, i.e., 2000. Based on $f(s, \theta)$, the further definitions of the performance of a PAP solver on an instance (used as the fitness function in the evolution of the instance population), and the performance of a solver on an instance set (used for fitness evaluation to compare different PAPs) are analogous to the case of TSP.

V. COMPUTATIONAL STUDIES

To assess the potential of CEPS, computational studies have been conducted with CEPS-TSP and CEPS-VRPSPDTW.¹ The experiments mainly aim to address two questions.

- 1) Whether CEPS could better tackle the few-shots challenge, i.e., build generalizable PAPs with limited training instances, compared with the state-of-the-art PAP construction methods.
- 2) Whether co-evolution, i.e., alternately evolving the configuration population and the instance population, is

effective as expected at enhancing the generalization of the resultant PAPs.

To answer these two questions, two instance sets were firstly generated for TSP and VRPSPDTW, respectively. The TSP instance set consists of 500 instances and the VRPSPDTW instance set consists of 233 instances. It should be noted that these instances are generated as our testbed. To avoid bias toward CEPS, these instances should not be generated in the same way that CEPS evolves the instance population. After the benchmark sets were generated, each of them was then randomly split into a training and a testing set, the size of which is 6% and 94% of the whole set, respectively. To reduce the effect of the random splitting on the experimental results, the split was repeated for three times, leading to three unique pairs of training and testing sets for TSP and VRPSPDTW, denoted as TSP_1/2/3 and VRPSPDTW_1/2/3, respectively. Throughout the experiments, testing instances were only used to approximate the generalization performance of the PAPs obtained by CEPS and compared methods. Only the training instances were used for PAP construction, regardless of the methods used. The TSP/VRPSPDTW instance set, the compared methods, and experimental protocol are further elaborated as follows.

A. Benchmark Instances

For TSP, we collected ten different instance generators from the literature, namely, *portgen*, *ClusteredNetwork*, *explosion*, *implosion*, *cluster*, *rotation*, *linearprojection*, *expansion*, *compression*, and *gridmutation*. Among them, *portgen* generates a TSP instance (called *rue* instance) by uniformly randomly placing the points on a Euclidean plane. It has been used to create test beds for the 8th DIMACS Implementation Challenge [35]. The generator *ClusteredNetwork* is from the R-package *netgen* [36], which generates an instance by placing points around different central points. The other eight generators are proposed by a recent study [37], which generate a TSP instance mainly by simulating a phenomenon in the point clouds of a *rue* instance. The details of these generators could be found in Appendix A. Considering the rather different instance-generation mechanisms underlying them, they are expected to generate highly diverse TSP instances. We used each of them to generate 50 instances, which finally gave us a set of 500 TSP instances. The problem sizes (i.e., city number) of all these instances are 800.

For VRPSPDTW, we obtained data from a real-world application of the JD logistics company. Specifically, the data contain customer requests that occurred during a period of time in a city. The total number of customers is 3000, of which 400 customers require service per day. Therefore, to generate a VRPSPDTW instance, we randomly select 400 customers from the 3000 customers, and the pickup/delivery demands of each customer are randomly selected from all the demands that the customer has during this period of time. We repeated this process for 500 times, thus obtaining a set of 500 VRPSPDTW instances. After that, a VRPSPDTW solver [18] was used to determine whether the generated instances have feasible solutions and those without feasible solutions were discarded. Finally, we obtained a set of 233 VRPSPDTW instances.

¹The source code of CEPS-TSP and CEPS-VRPSPDTW, as well as the benchmark instances generated for the experiments, have been made available at <https://github.com/senshineL/CEPS>.

TABLE I
SUMMARY OF THE EXPERIMENTAL SETTINGS

	Instance Sets	#solvers in PAP	Performance Indicator	Parameterized Algorithm
TSP	500 instances generated by 10 different generators. Training/Testing split: 30/470	4	Runtime needed to find the optima of the instances. In particular, PAR-10 with cutoff time 10 seconds was used (see Section IV-A)	LKH version 2.0.7 [34] with 23 parameters
VRPSPDTW	233 instances obtained from real-world application. Training/Testing split: 14/219	4	Cost of the found solutions. In particular, PANC with cutofftime 150 seconds was used (see Section IV-B)	Co-GA [18] (with 12 parameters)

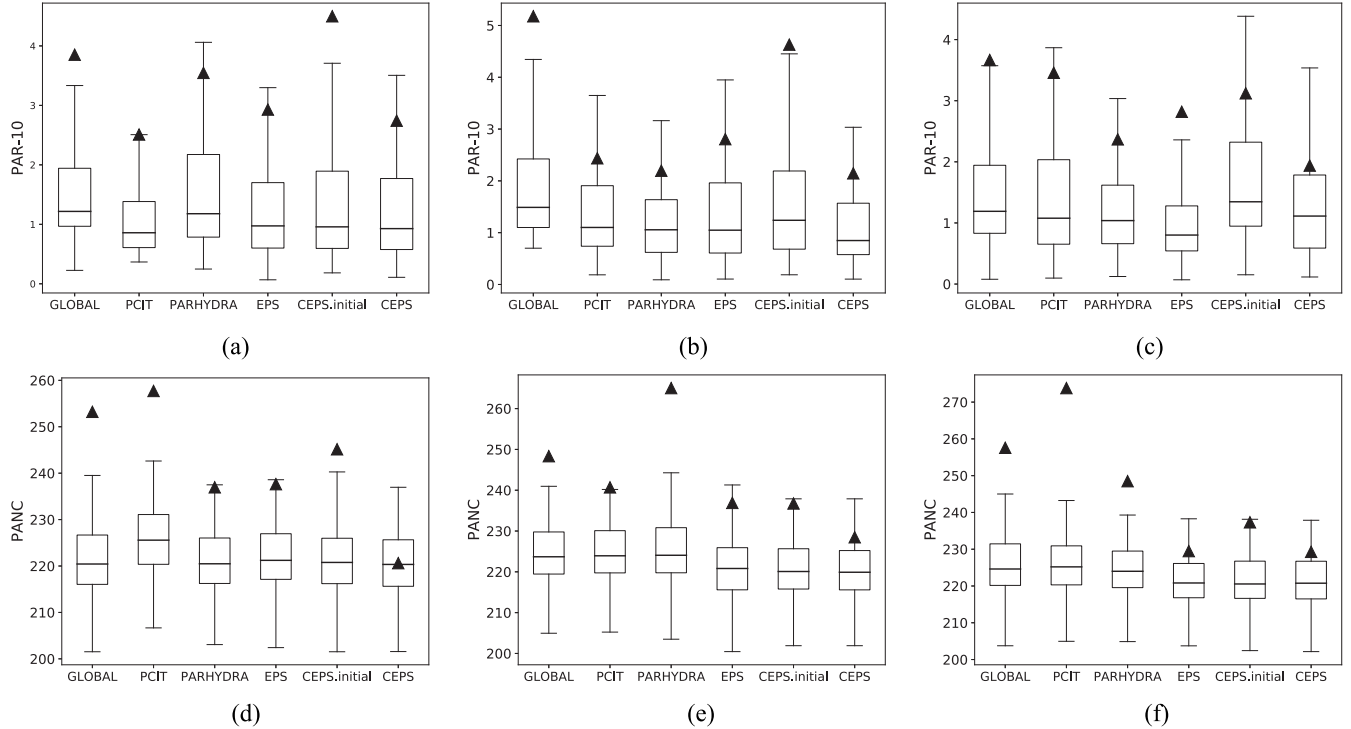


Fig. 1. Visual comparison in boxplots of the medians and variance of the test performance of each PAP across the testing instances. Note the mean value is also plotted, indicated by “▲.” (a) TSP-1. (b) TSP-2. (c) TSP-3. (d) VRPSPDTW-1. (e) VRPSPDTW-2. (f) VRPSPDTW-3.

B. Compared Methods

We compared CEPS with the state-of-the-art PAP construction methods (see Section II-B), namely, GLOBAL [25], PARHYDRA [25], [26], and PCIT [13]. It should be noted that all these methods involve no instance generation mechanism, i.e., the given training instances are assumed to sufficiently represent the target problem class. Hence, given our experimental settings, comparison between CEPS and these approaches aims to evaluate whether CEPS could better tackle the few-shots challenge.

To address research question 2) raised at the beginning of Section V, i.e., the role of co-evolution for achieving better (if any) generalization, a baseline method, named evolution of parameterized search (EPS), was also adopted in the comparison. EPS differs from CEPS in that it conducts instance mutation and PAP construction in two isolated phases, rather than alternately. Given the same training instance set as CEPS, EPS first applies the instance mutation operator to generate an augmented set of instances. The size of this augmented set is kept the same as the number of instances generated during the whole procedure of CEPS. Then, a PAP is evolved with the same approach as in CEPS, but using the union of the initial and the augmented training instance sets as

the input. Moreover, to further verify the effectiveness of the co-evolution in CEPS, we included the initial PAPs of CEPS (the PAPs built by the initialization phase, lines 2–7 in Algorithm 1) in the comparison with the final PAPs obtained by CEPS.

C. Experimental Protocol

We set the number of component solvers in PAP, i.e., K , to 4, since 4-core machines are widely available now. The parameters of the compared methods were set following suggestions in the literature. For CEPS, the number of iterations of the co-evolution, i.e., $MaxIte$, and the number of temporary PAPs generated, i.e., n , were set to 4 and 10, respectively. The termination condition for the evolution of the instance population in CEPS was the predefined time budget being exhausted. In the experiments, the total CPU time consumed by each compared method was kept almost the same. The detailed time settings of each compared method are presented in Appendix B. The above-described experimental settings, as well as the used parameterized algorithms and performance indicators (see Section IV), are summarized in Table I.

For each pair of training and testing sets, i.e., TSP_1/2/3 and VRPSPDTW_1/2/3, we applied each considered method

TABLE II
 TESTING RESULTS OF THE PAPs CONSTRUCTED BY EACH METHOD. #TOS REFER TO THE NUMBER OF TOTAL TIMEOUTS. PAR-10 AND PANC ARE PENALIZED AVERAGE RUNTIME-10 AND PANC, RESPECTIVELY. PERFORMANCE OF A PAP IS HIGHLIGHTED IN GRAY IF IT ACHIEVED THE BEST TESTING PERFORMANCE

	TSP-1		TSP-2		TSP-3		VRPSPDTW-1		VRPSPDTW-2		VRPSPDTW-3	
	#TOS	PAR-10	#TOS	PAR-10	#TOS	PAR-10	#TOS	PANC	#TOS	PANC	#TOS	PANC
GLOBAL	10	3.85	15	5.18	10	3.67	4	253	3	248	4	258
PCIT	6	2.51	4	2.44	9	3.46	4	258	2	240	6	274
PARHYDRA	9	3.55	4	2.19	5	2.36	2	237	5	265	3	249
EPS	7	2.93	6	2.81	8	2.81	2	238	2	236	1	229
CEPS.initial	14	4.50	14	4.63	6	3.12	3	245	2	237	2	237
CEPS	6	2.74	4	2.15	2	1.94	0	221	1	229	1	229

to construct a PAP on the training set and then tested the resulting PAP on the testing set. For each testing instance, the PAP was applied for three runs and the median of those three runs was recorded as the performance of the solver on the instance. The performance of a PAP on different testing instances were then aggregated to obtain the number of total timeouts (#TOS), PAR-10 (for TSP solver), and PANC (for VRPSPDTW solver) on the testing set. All the experiments were conducted on a cluster of 3 Intel Xeon machines with 128-GB RAM and 24 cores each (2.20 GHz, 30-MB Cache), running Centos 7.5.

D. Results and Analysis

We report the #TOS, PAR-10, and PANC achieved by the PAPs on the testing set in Table II and also visualize their medians and variance across all the testing instances by boxplots in Fig. 1. Note the mean value is also plotted in Fig. 1 (indicated by “▲”) to show that for a PAP how its PAR-10/PANC is affected by the outliers (the timeout cases), which would be hidden by boxplots. In Table II #TOS, PAR-10/PANC of a PAP is highlighted in gray if it achieved the best performance. One could make three important observations from these results. First, the PAPs obtained by CEPS have the smallest number of timeouts in all the six experiments, which means they have the highest success rate for solving the testing instances among all the tested PAPs. Recall that CEPS actively searches in the instance space to identify the hard-to-solve instances for further improving the generalization of PAPs. Such a mechanism makes CEPS the method that is least affected by the hard testing instances, which significantly differs from the given training instances. This could be further verified by Fig. 1, in which CEPS is the method that has the least gap between the mean value (which takes timeouts into account) and median value (which naturally filters out the timeouts). Moreover, thanks to the least number of timeouts, in five out of the six experiments, the PAPs output by CEPS achieved the best scores in terms of PAR-10 and PANC. Typically, one could observe that in TSP-1 and TSP-3 of Fig. 1, PARHYDRA and EPS have “better” performances than CEPS on the normal instances, but finally achieved worse PAR-10 due to more timeouts. Furthermore, recall that in different experiments, the training/testing splits were different, compared to other approaches, CEPS performed more stably over all six experiments. For instance, the #TOS of PCIT and PARHYDRA fluctuate over different training/testing sets

on VRPSPDTW problem. In summary, CEPS is not only the best-performing method but also is less sensitive to the training data, i.e., could better tackle the few-shots challenge.

Second, EPS also involves instance generation, while was outperformed by methods that do not generate synthetic instances in several cases, e.g., compared to PARHYDRA on TSP_2. This observation indicates that isolating instance generation from PAP construction may have negative effects. On the other hand, the fact CEPS performed better than EPS shows the effectiveness of integrating instance generation into the co-evolving framework.

Third, compared to the initial PAPs of CEPS (indicated by “CEPS.initial” in Table II), the final PAPs obtained by CEPS performed better on all of the six experiments. On average, the performance improvement rate (in terms of PAR-10 and PANC) is 21.78%. These results indicate that the co-evolution in CEPS is effective as expected at enhancing the generalization of the PAP solvers.

E. Comparison With State-of-the-Art TSP Solvers

For CEPS-TSP, to further assess its performance, we compared the PAPs constructed by it with the state-of-the-art TSP solvers. More specifically, we considered the following.

- 1) The default configuration of the considered LKH algorithm, denoted as LKH-default.
- 2) The default configuration of another powerful TSP algorithm, EAX [38], denoted as EAX-default, which has been demonstrated to outperform LKH on a broad range of TSP instances.
- 3) The tuned versions of LKH and EAX, denoted as LKH-tuned and EAX-tuned, respectively, which are obtained by running SMAC on their configuration spaces and the training sets for the same time budget of the PAP construction of CEPS.

In addition, we also considered two state-of-the-art portfolio-based AS methods for TSP [29], [30] (see Section II-B). Since these two methods have adopted the same TSP algorithm portfolio, which contains LKH, EAX, MAOS [39], and their variants, instead of comparing each of these two method with CEPS, we directly adopted the virtual best solver (VBS) of their algorithm portfolio. VBS is the *oracle* or *perfect* selector that always chooses the best algorithm for each instance without any selecting cost. This idealized procedure provides an upper bound for the performance of any algorithm selector; due to imperfect selection and the cost incurred

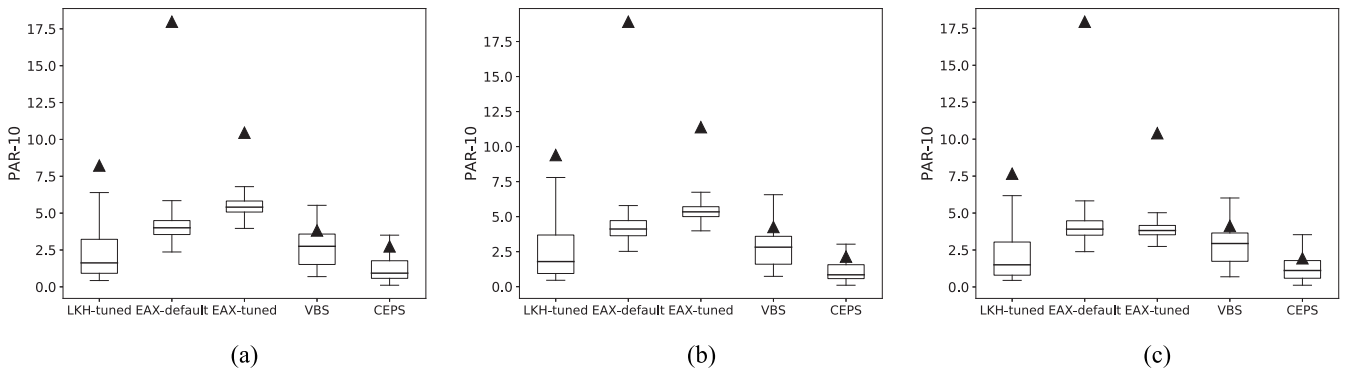


Fig. 2. Visual comparison in boxplots of the medians and variance of the test performance of each TSP solver across the testing instances. Note the mean value is also plotted, indicated by “▲”. (a) TSP-1. (b) TSP-2. (c) TSP-3.

TABLE III
COMPARISON OF THE STATE-OF-THE-ART TSP SOLVERS WITH THE PAPS OBTAINED BY CEPS, ON THE TESTING SET. #TOS REFER TO THE NUMBER OF TOTAL TIMEOUTS. PAR-10 IS THE PENALIZED AVERAGE RUNTIME-10. PERFORMANCE OF A SOLVER IS HIGHLIGHTED IN GRAY IF IT ACHIEVED THE BEST TESTING PERFORMANCE

	TSP-1		TSP-2		TSP-3	
	#TOS	PAR-10	#TOS	PAR-10	#TOS	PAR-10
LKH-default	131	30.84	137	31.98	150	34.73
LKH-tuned	29	8.23	34	9.40	27	7.67
EAX-default	69	17.98	73	18.91	69	17.95
EAX-tuned	33	10.97	30	10.38	29	10.12
VBS	6	3.82	7	4.26	6	4.13
CEPS	6	2.74	4	2.15	2	1.94

by selecting, VBS cannot be achieved in practice by actual algorithm selectors.

As before, for each testing instance, we applied each solver for three runs and the median of those three runs was recorded as the performance of the solver on the instance. We report the #TOS and PAR-10 achieved by these solvers on the testing set in Table III and also visualize their medians and variance across all the testing instances by boxplots in Fig. 2. Note LKH-default is omitted in Fig. 2 due to its large number of timeouts. There are two important findings from these results. First, LKH-default and EAX-default performed badly on the testing set, with considerable timeouts. We speculate that this is because the default configurations of LKH and EAX are mainly designed to handle much larger scale TSP instances (e.g., 10 000) than the instances considered here. After being tuned (i.e., LKH-tuned and EAX-tuned), they both achieved significant performance improvement, though still obviously falling behind of the PAPS obtained by CEPS. Second, the only solver that could match the PAP’s performance level in one of the three scenarios is the VBS of the algorithm portfolio considered by the AS approaches [29], [30]. However, in TSP-2 and TSP-3, the performance advantage of the PAP is still significant.

F. Assessing Generalization on Existing VRPSPDTW Benchmarks

To further investigate the generalization ability of CEPS, the PAP constructed by CEPS in the VRPSPDTW_1 experiment has been tested on existing VRPSPDTW benchmarks [18],

TABLE IV
COMPARISON BETWEEN THE SOLUTIONS FOUND BY PAP AND THE BEST-KNOWN SOLUTIONS (BKSS) FOUND BEFORE (AS REPORTED IN THE LITERATURE) ON EXISTING VRPSPDTW BENCHMARK INSTANCES. #BETTER, #NOT-WORSE, AND #WORSE REFER TO THE NUMBER OF THE INSTANCES ON WHICH PAP FOUND BETTER, NOT WORSE (I.E., EITHER WITH BETTER OR THE SAME QUALITY), AND WORSE SOLUTIONS COMPARED TO BKSS

Instance Type	#instances	#better	#not-worse	#worse
RCdp (small)	9	0 (0%)	9 (100%)	0 (0%)
Rdp	23	4 (17%)	17 (57%)	6 (26%)
Cdp	17	0 (0%)	9 (53%)	8 (47%)
RCdp	16	6 (38%)	8 (38%)	8 (50%)
count	65	10 (15%)	43 (66%)	22 (34%)

which is a widely used benchmark for VRPSPDTW. Note that compared to the benchmarks, the training instances in VRPSPDTW_1 were obtained from different sources (i.e., real-world application), and may have quite different problem characteristics, e.g., customer number and node distribution. Hence, the PAP constructed by CEPS could be said to generalize well to totally unseen data if it was constructed using the VRPSPDTW_1 training set while still performing well on the VRPSPDTW benchmarks. Table IV presents the comparisons between the solutions found by the PAP and the best-known solutions reported in the literature [18], [40]–[43] (up to May 2019), regardless of what algorithm was used. Table III shows that overall the PAP could generalize well to the existing benchmarks. On 43 out of 65 (66%) instances, the solutions found by the PAP are not worse than the best solutions currently known. It is notable that on ten instances, the PAP found new best solutions. Another observation is that the PAP performed not very well on the “cdp” type instances, in which the locations of customers are clustered (see [18] for details). We speculate that this is because the parameterized algorithm used for tuning PAPS has an inherent deficiency when handling this type of instances, which on the other hand indicates that highly parameterized algorithms with flexible solving capacities are important to fully exploit the power of CEPS on a specific problem class.

VI. THREATS TO VALIDITY

There are several validity threats to the findings of this study. The first one is the correctness of the implementation of all

the compared methods. Prior to commencing our experiments, we have thoroughly checked the source code of these methods (obtained from online or implemented by ourselves) and ensured that the implementations were correct.

Second, the results of our experiments are limited to the data sets used, in which the TSP instances are generated by ten different generators while the VRPSPDTW instances are collected from real-world application. We have made these instances available online. In the future, we will assess CEPS-TSP and CEPS-VRPSPDTW on more instance sets obtained from other sources (generators and applications).

Third, although we have demonstrated that CEPS could better tackle the few-shots challenge than existing PAP construction methods in two case studies, there is no guarantee that CEPS could be easily applied to other problem domains. Actually, an instantiation of CEPS to a specific domain involves specification of instance mutation operator, the parameterized algorithm, as well as the automatic algorithm configuration method. Hence, as demonstrated by CEPS-TSP and CEPS-VRPSPDTW, one needs to first define these three modules according to previous literature on the target problem class, or from scratch. Among these three modules, the instance generators could be adapted from one problem to another more easily and the automatic algorithm configuration methods are usually generic. Hence, the parameterized search method might be the most crucial (also the most difficult-to-obtain) one among the three modules.

VII. CONCLUSION

In this work, a co-evolutionary approach, i.e., CEPS, is proposed for constructing PAPs to obtain good generalization performance. By co-evolving the training instance set and the configurations, CEPS gradually guides the search of configurations towards instances on which the current configurations fail to perform well, and thus leads to PAPs that could generalize better. From a theoretical point of view, the evolution of instance set is essentially a greedy mechanism, for instance, augmentation that guarantees the generalization performance of the resultant solver to improve as much as possible. As a result, CEPS is particularly effective in case that only a limited number of problem instances is available. Such a scenario is usually true when building real-world systems for tackling hard optimization problems. Two concrete instantiations, i.e., CEPS-TSP and CEPS-VRPSPDTW, are also presented. The performance of the two instantiations on TSP and VRPSPDTW problems support the effectiveness of CEPS in the sense that in comparison with state-of-the-art PAP construction approaches, the PAPs obtained by CEPS achieve better generalization performance.

Since CEPS is a generic framework, some discussions would help elaborate issues that are of significance in practice. First, although this work assumes CEPS takes a set of initial training instances as the input, such training instances are not necessarily real-world instances but could be generated randomly. In other words, CEPS could be used in a fully cold-start setting (a.k.a. zero shot), i.e., no real-world instances are available for the target problem class. Furthermore, CEPS could either be run offline or online, i.e., it could accommodate new real instances whenever available.

TABLE V
DETAILED TIME SETTINGS (IN HOURS) OF EACH
PAP CONSTRUCTION METHOD

	TSP				CPU Time
	t_c	t_v	t_i	t_{ini}	
CEPS	1.5h	0.5h	1.5h	8h	320h
GLOBAL	7.5h	1h	–	–	340h
PCIT	7.5h	1h	–	–	340h
PARHYDRA	2h	1h	–	–	300h
	VRPSPDTW				CPU Time
	t_c	t_v	t_i	t_{ini}	
CEPS	6h	2h	6h	32h	1312h
GLOBAL	30h	4h	–	–	1360h
PCIT	30h	4h	–	–	1360h
PARHYDRA	8h	4h	–	–	1200h

Second, the potential of CEPS could be further explored by taking advantage of the data generated during its run, except for the final obtained PAP. The data contain all the sampled configurations and instances, and the performance of the former on the latter. Considering that when using a search method to solve a problem instance, its optimal parameter values are usually problem-instance dependent and thus need to be tuned. To tune parameters for a new problem instance, we can learn from the historical data generated by CEPS to build a mapping from problem instances to their optimal parameter values, i.e., a low-cost online parameter-tuning system for any single instance. It could be seen as an extension of the common AS systems [31], which select the best algorithm/configuration for a given instance from a predefined algorithm set. In addition, the challenging instance sets generated by CEPS could be further used on comprehensive analysis of the strengths and weaknesses of the parameterized algorithms, such as for which configurations are those instances challenging and further improvement of the algorithm. We have made the instance sets generated by CEPS available online to further facilitate the investigations on them.

Finally, it is interesting to note that the emerging topic of learn to optimize, which explores utilizing machine learning techniques, e.g., reinforcement learning, to build neural networks for solving optimization problems [19]–[21], [44], [45], could also be combined with CEPS. In this case, the implementation of CEPS would be able to leverage on gradient descent methods to tune/evolve the configurations (i.e., training the weights of a network).

APPENDIX A TSP INSTANCE GENERATORS

The adopted ten TSP generators include the *portgen* generator from the 8th DIMACS Implementation Challenge [35], the *ClusteredNetwork* generator from the R-package *netgen* [36], and eight TSP instance generators, namely, explosion, implosion, cluster, rotation, linearprojection, expansion, compression, and gridmutation, from the R-package *tspgen* [37].

- 1) The *portgen* generator generates an instance by uniformly randomly placing the points. The generated instances are called *rue* instances.
- 2) The *ClusteredNetwork* generator generates an instance by placing points around different central points. The number of the clusters was set to 4, 5, 6, 7, and 8, for each of which ten instances were generated.

- 3) The *explosion* generator generates an instance by tearing holes into the city points of a *rue* instance, with all points within the explosion range pushed out of the explosion area.
- 4) The *implosion* generator generates an instance by driving the city points of a *rue* instance toward a randomly sampled implosion center.
- 5) The *cluster* generator generates an instance by randomly sampling a cluster centroid in a *rue* instance, and then moving a randomly selected set of points into the cluster region.
- 6) The *rotation* generator generates an instance by rotating a subset of points of a *rue* instance with a randomly selected angle.
- 7) The *linearprojection* generator generates an instance by projecting a subset of points of a *rue* instance to a linear function.
- 8) The *expansion* generator generates an instance by placing a tube around a linear function in the points of a *rue* instance, and then orthogonally pushes all points within that tube out of that region.
- 9) The *compression* generator generates an instance by squeezing a set of randomly selected points of a *rue* instance from within a tube (surrounding a linear function) toward the tube's central axis.
- 10) The *gridmutation* generator generates an instance by randomly relocating a "box" of city points of a *rue* instance.

APPENDIX B

DETAILED TIME SETTINGS OF COMPARED METHODS

The most time-consuming parts of PAP construction methods are the runs of the configurations on the problem instances, and the incurred computational costs account for the vast majority of the total costs. For CEPS, the configurations would be run in the initialization phase (line 5 in Algorithm 1), in the evolution of the configuration population (lines 12 and 15 in Algorithm 1) and in the evolution of the instance population (line 22 in Algorithm 1). Therefore, for each of these four procedures we set the corresponding wall-clock time budget, i.e., t_{init} , t_c , t_v , and t_i , to control the overall computational costs of CEPS. Then, the total CPU time consumed by CEPS could be estimated by $t_{\text{init}} + \text{MaxIte} \cdot K \cdot [n \cdot (t_c + t_v) + t_i]$. In this article, K , MaxIte , and n are set to 4, 4, and 10, respectively.

The total CPU time consumed by GLOBAL and PCIT could be estimated by $K \cdot n \cdot (t_c + t_v)$, while for PARHYDRA it could be estimated by $\sum_{i=1}^K i \cdot n \cdot (t_c + t_v)$ (see [13], [25], and [26] for how these results are derived). Note for different methods t_c , t_v , and t_i could be set to different values. The detailed setting of the time budget for each PAP construction method is given in Table V. Overall, the total CPU time consumed by each method is kept almost the same.

REFERENCES

- [1] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman Problem," *Oper. Res.*, vol. 21, no. 2, pp. 498–516, 1973.
- [2] A. E. Eiben and S. K. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms," *Swarm Evol. Comput.*, vol. 1, no. 1, pp. 19–31, 2011.
- [3] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Proc. 5th Int. Conf. Learn. Intell. Optim. (LION)*, Rome, Italy, Jan. 2011, pp. 507–523.
- [4] G. Karafotias, M. Hoogendoorn, and Á. E. Eiben, "Parameter control in evolutionary algorithms: Trends and challenges," *IEEE Trans. Evol. Comput.*, vol. 19, no. 2, pp. 167–187, Apr. 2015.
- [5] C. Huang, Y. Li, and X. Yao, "A survey of automatic parameter tuning methods for metaheuristics," *IEEE Trans. Evol. Comput.*, vol. 24, no. 2, pp. 201–216, Apr. 2020.
- [6] A. S. D. Dymond, A. P. Engelbrecht, S. Kok, and P. S. Heyns, "Tuning optimization algorithms under multiple objective function evaluation budgets," *IEEE Trans. Evol. Comput.*, vol. 19, no. 3, pp. 341–358, Jun. 2015.
- [7] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: An automatic algorithm configuration framework," *J. Artif. Intell. Res.*, vol. 36, no. 1, pp. 267–306, 2009.
- [8] C. Ansótegui, M. Sellmann, and K. Tierney, "A gender-based genetic algorithm for the automatic configuration of algorithms," in *Proc. 15th Int. Conf. Principles Pract. Constraint Program. (CP)*, Lisbon, Portugal, Sep. 2009, pp. 142–157.
- [9] C. Ansótegui, Y. Malitsky, H. Samulowitz, M. Sellmann, and K. Tierney, "Model-based genetic algorithms for algorithm configuration," in *Proc. 24th Int. Joint Conf. Artif. Intell. (IJCAI)*, Buenos Aires, Argentina, Jul. 2015, pp. 733–739.
- [10] M. López-Ibañez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari, "The irace package: Iterated racing for automatic algorithm configuration," *Oper. Res. Perspect.*, vol. 3, pp. 43–58, Sep. 2016.
- [11] C. P. Gomes and B. Selman, "Algorithm portfolios," *Artif. Intell.*, vol. 126, nos. 1–2, pp. 43–62, 2001.
- [12] B. A. Huberman, R. M. Lukose, and T. Hogg, "An economics approach to hard computational problems," *Science*, vol. 275, no. 5296, pp. 51–54, 1997.
- [13] S. Liu, K. Tang, and X. Yao, "Automatic construction of parallel portfolios via explicit instance grouping," in *Proc. 33rd AAAI Conf. Artif. Intell. (AAAI)*, Honolulu, HI, USA, Jan. 2019, pp. 1560–1567.
- [14] F. Peng, K. Tang, G. Chen, and X. Yao, "Population-based algorithm portfolios for numerical optimization," *IEEE Trans. Evol. Comput.*, vol. 14, no. 5, pp. 782–800, Oct. 2010.
- [15] S. Liu, K. Tang, and X. Yao, "Generative adversarial construction of parallel portfolios," *IEEE Trans. Cybern., early access*, Apr. 29, 2020, doi: [10.1109/TCYB.2020.2984546](https://doi.org/10.1109/TCYB.2020.2984546).
- [16] K. Asanovic *et al.*, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, 2009.
- [17] G. Reinelt, "TSPLIB—A traveling salesman problem library," *INFORMS J. Comput.*, vol. 3, no. 4, pp. 376–384, 1991.
- [18] H.-F. Wang and Y.-Y. Chen, "A genetic algorithm for the simultaneous delivery and pickup problems with time window," *Comput. Ind. Eng.*, vol. 62, no. 1, pp. 84–95, 2012.
- [19] M. Nazari, A. Oroojlooy, L. V. Snyder, and M. Takác, "Reinforcement learning for solving the vehicle routing problem," in *Proc. 31st Annu. Conf. Neural Inf. Process. Syst. (NeurIPS)*, Dec. 2018, pp. 9861–9871.
- [20] X. Chen and Y. Tian, "Learning to perform local rewriting for combinatorial optimization," in *Proc. 32nd Annu. Conf. Neural Inf. Process. Syst. (NeurIPS)*, Dec. 2019, pp. 6278–6289.
- [21] W. Kool, H. van Hoof, and M. Welling, "Attention, learn to solve routing problems!," 2018. [Online]. Available: [arXiv:1803.08475](https://arxiv.org/abs/1803.08475)
- [22] A. Blot, H. H. Hoos, L. Jourdan, M.-É. Kessaci-Marmion, and H. Trautmann, "MO-ParamILS: A multi-objective automatic algorithm configuration framework," in *Proc. 10th Int. Conf. Learn. Intell. Optim. (LION)*, Ischia, Italy, Jun. 2016, pp. 32–47.
- [23] M. Birattari, "On the estimation of the expected performance of a meta-heuristic on a class of instances. How many instances, how many runs?" Dept. Comp. & Decis. Eng., Université Libre de Bruxelles, Brussels, Belgium, Rep. TR/IRIDIA/2004-01, 2004. [Online]. Available: <https://code.ulb.ac.be/>
- [24] S. Liu, K. Tang, Y. Lei, and X. Yao, "On performance estimation in automatic algorithm configuration," in *Proc. 34th AAAI Conf. Artif. Intell.*, Feb. 2020, pp. 2384–2391.
- [25] M. Lindauer, H. H. Hoos, K. Leyton-Brown, and T. Schaub, "Automatic construction of parallel portfolios via algorithm configuration," *Artif. Intell.*, vol. 244, pp. 272–290, Mar. 2017.
- [26] L. Xu, H. H. Hoos, and K. Leyton-Brown, "Hydra: Automatically configuring algorithms for portfolio-based selection," in *Proc. 24th AAAI Conf. Artif. Intell.*, Jul. 2010, pp. 210–216.
- [27] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney, "ISAC—Instance-specific algorithm configuration," in *Proc. 19th Eur. Conf. Artif. Intell. (ECAI)*, Aug. 2010, pp. 751–756.
- [28] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," *J. Artif. Intell. Res.*, vol. 32, pp. 565–606, Jun. 2008.

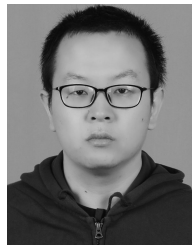
- [29] P. Kerschke, L. Kotthoff, J. Bossek, H. H. Hoos, and H. Trautmann, "Leveraging TSP solver complementarity through machine learning," *Evol. Comput.*, vol. 26, no. 4, pp. 597–620, 2018.
- [30] K. Zhao, S. Liu, Y. Rong, and J. X. Yu, "Leveraging TSP solver complementarity via deep learning," 2020. [Online]. Available: arXiv:2006.00715
- [31] L. Kotthoff, "Algorithm selection for combinatorial search problems: A survey," *AI Mag.*, vol. 35, no. 3, pp. 48–60, 2014.
- [32] C. D. Rosin and R. K. Belew, "New methods for competitive coevolution," *Evol. Comput.*, vol. 5, no. 1, pp. 1–29, 1997.
- [33] J. I. van Hemert, "Evolving combinatorial problem instances that are difficult to solve," *Evol. Comput.*, vol. 14, no. 4, pp. 433–462, Dec. 2006.
- [34] K. Helsingaun, "General k -opt submoves for the Lin-Kernighan TSP heuristic," *Math. Program. Comput.*, vol. 1, nos. 2–3, pp. 119–163, 2009.
- [35] D. S. Johnson and L. A. McGeoch, "Experimental analysis of heuristics for the STSP," in *The Traveling Salesman Problem and Its Variations*, G. Gutin and A. P. Punnen, Eds. Boston, MA, USA: Springer, 2007, pp. 369–443.
- [36] J. Bossek. (2015). *netgen: Network Generator for Combinatorial Graph Problems*. [Online]. Available: <https://github.com/jakobbossek/netgen>
- [37] J. Bossek, P. Kerschke, A. Neumann, M. Wagner, F. Neumann, and H. Trautmann, "Evolving diverse TSP instances by means of novel and creative mutation operators," in *Proc. 15th ACM/SIGEVO Conf. Found. Genet. Algorithms (FOGA)*, Aug. 2019, pp. 58–71.
- [38] 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.
- [39] X.-F. Xie and J. Liu, "Multiagent optimization system for solving the traveling salesman problem (TSP)," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 39, no. 2, pp. 489–502, Apr. 2009.
- [40] C. Wang, D. Mu, F. Zhao, and J. W. Sutherland, "A parallel simulated annealing method for the vehicle routing problem with simultaneous pickup–delivery and time windows," *Comput. Ind. Eng.*, vol. 83, pp. 111–122, May 2015.
- [41] W. Huang and T. Zhang, "Vehicle routing problem with simultaneous pick-up and delivery and time-windows based on improved global artificial fish swarm algorithm," *Comput. Eng. Appl.*, vol. 52, no. 21, pp. 21–29, 2016.
- [42] X. Pu and K. Wang, "An evolutionary ant colony algorithm for a vehicle routing problem with simultaneous pick-up and delivery and hard time windows," in *Proc. 30th Chin. Control Decis. Conf. (CCDC)*, Shenyang, China, Jun. 2018, pp. 6499–6503.
- [43] Y. Shi, T. Boudouh, and O. Grunder, "An efficient Tabu search based procedure for simultaneous delivery and pick-up problem with time window," *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 241–246, 2018.
- [44] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [45] T. D. Kulkarni, K. R. Narasimhan, A. Saeedi, and J. B. Tenenbaum, "Hierarchical deep reinforcement learning: integrating temporal abstraction and intrinsic motivation," in *Proc. 29th Annu. Conf. Neural Inf. Process. Syst. (NIPS)*, Dec. 2016, pp. 3675–3683.



Ke Tang (Senior Member, IEEE) received the B.Eng. degree from the Huazhong University of Science and Technology, Wuhan, China, in 2002, and the Ph.D. degree from Nanyang Technological University, Singapore, in 2007.

From 2007 to 2017, he was with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China, first as an Associate Professor from 2007 to 2011 and later as a Professor from 2011 to 2017. He is currently a Professor with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China. He has published more than 160 papers, which have received over 9000 Google Scholar citations with an H-index of 45. His major research interests include evolutionary computation and machine learning, particularly in large-scale evolutionary computation, integration of evolutionary computation and machine learning, as well as their applications.

Prof. Tang was a recipient of the Royal Society Newton Advanced Fellowship in 2015 and the IEEE Computational Intelligence Society Outstanding Early Career Award in 2018. He is an Associate Editor of the IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION and served as a member of Editorial Boards for a few other journals.



Shengcai Liu (Member, IEEE) received the B.Eng. and Ph.D. degrees in computer science and technology from the University of Science and Technology of China, Hefei, China, in 2014 and 2020, respectively.

Since 2021, he has been working as a Research Assistant Professor with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China. His research interests include combinatorial optimization, automatic parameter tuning, and their applications.



Peng Yang (Member, IEEE) received the B.Eng. and Ph.D. degrees in computer science and technology from the University of Science and Technology of China, Hefei, China, in 2012 and 2017, respectively.

Since 2018, he has been working as a Research Assistant Professor with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China. His research interests include large-scale distributed evolutionary computation system and its applications.

Dr. Yang has been served as a regular reviewer of several world-class journals and a Program Committee Member of a set of top international conferences.



Xin Yao (Fellow, IEEE) received the B.Sc. degree from the University of Science and Technology of China (USTC), Hefei, China, in 1982, the M.Sc. degree from the North China Institute of Computing Technologies, Langfang, China, in 1985, and the Ph.D. degree from USTC in 1990.

He is a Chair Professor of Computer Science with the Southern University of Science and Technology, Shenzhen, China, and a part-time Professor of Computer Science with the University of Birmingham, Birmingham, U.K. His major

research interests include evolutionary computation, ensemble learning, and their applications to software engineering.

Dr. Yao's work won the 2001 IEEE Donald G. Fink Prize Paper Award; 2010, 2016, and 2017 IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION Outstanding Paper Awards; 2011 IEEE TRANSACTIONS ON NEURAL NETWORKS Outstanding Paper Award; and many other best paper awards at conferences. He received a prestigious Royal Society Wolfson Research Merit Award in 2012, the IEEE CIS Evolutionary Computation Pioneer Award in 2013, and the 2020 IEEE Frank Rosenblatt Award. He was the President of IEEE CIS from 2014 to 2015 and the Editor-in-Chief of IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION from 2003 to 2008. He is an IEEE Fellow and was a Distinguished Lecturer of the IEEE Computational Intelligence Society.