Generative Adversarial Construction of Parallel Portfolios

Shengcai Liu, *Student Member, IEEE*, Ke Tang, *Senior Member, IEEE*, and Xin Yao, *Fellow, IEEE*

***Abstract*—Since automatic algorithm configuration methods have been very effective, recently there is increasing research interest in utilizing them for automatic solver construction, resulting in several notable approaches. For these approaches, a basic assumption is that the given training set could sufficiently represent the target use cases such that the constructed solvers can generalize well. However, such an assumption does not always hold in practice since in some cases, we might only have scarce and biased training data. This article studies effective construction approaches for the parallel algorithm portfolios that are less affected in these cases. Unlike previous approaches, the proposed approach simultaneously considers instance generation and portfolio construction in an adversarial process, in which the aim of the former is to generate instances that are challenging for the current portfolio, while the aim of the latter is to find a new component solver for the portfolio to better solve the newly generated instances. Applied to two widely studied problem domains, that is, the Boolean satisfiability problems (SAT) and the traveling salesman problems (TSPs), the proposed approach identified parallel portfolios with much better generalization than the ones generated by the existing approaches when the training data were scarce and biased. Moreover, it was further demonstrated that the generated portfolios could even rival the state-of-the-art manually designed parallel solvers.**

***Index Terms*—Automatic portfolio construction (APC), generative adversarial approach, parallel algorithm portfolio, parameter tuning.**

I. Introduction

MANY high-performance algorithms for solving computationally hard problems, ranging from the exact methods such as mixed-integer programming solvers to heuristic methods, such as local search and metaheuristics, involve a large number of free parameters that need to be carefully tuned to achieve their best performance [1]–[4]. In many cases, finding performance-optimizing parameter settings is performed manually in an *ad-hoc* way. However, the manually tuning approach has two main disadvantages [5]–[8]: 1) it requires considerable human effort and 2) it is often limited to the exploration of few parameter settings, thus leading to a performance that is far from the optimal. As a result, there have been a lot of attempts on automated parameter tuning (see [6] for a comprehensive review), which is usually referred to as automatic algorithm configuration (AAC) [9]. Here, a configuration of a parameterized algorithm refers to a complete setting of the parameters of the algorithm such that the algorithm's behavior on a given problem instance is completely specified (up to randomization of the algorithm itself). In the last few years, with several high-performance algorithm configurators (i.e., AAC methods), such as ParamILS [6], GGA [10], irace [8], and SMAC [11] being proposed, AAC has become very effective.

As a consequence, recently there is increasing research interest in utilizing these methods to automatically construct effective solvers for a given application. The key idea is to parameterize many aspects of the algorithms and thus come up with a large space of algorithms as the configuration space, from which effective algorithm configurators are used to identify the high-performance algorithms. Unlike manual solver-designing paradigm which usually relies on considerable effort by human experts, the automatic solver construction approaches involve much less human effort and instead usually need to consume large budgets of computational time for configuration. This is acceptable (and even appealing) since the available computing power has been rapidly becoming much cheaper than before.[1] Indeed, such approaches have been demonstrated to be both practical and effective in cases of constructing sequential solvers [12], [13]; sequential portfolios [14]–[16] (i.e., algorithm portfolios with selectors/scheduling); and parallel portfolios [17], [18].

Generally all these approaches require that a training set (i.e., a set of problem instances of the problem domain of interest) is available for constructing the solvers, particularly to evaluate the solvers in the construction process. Moreover,

Manuscript received May 17, 2019; revised October 5, 2019 and March 4, 2020; accepted March 23, 2020. Date of publication April 29, 2020; date of current version February 16, 2022. This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB1003102, in part by the Guangdong Provincial Key Laboratory under Grant 2020B121201001, in part by the Natural Science Foundation of China under Grant 61672478, in part by the Program for Guangdong Introducing Innovative and Enterpreneurial Teams under Grant 2017ZT07X386, in part by the Shenzhen Peacock Plan under Grant KQTD2016112514355531, and in part by the National Leading Youth Talent Support Program of China. This article was recommended by Associate Editor P. P. Angelov. *(Corresponding author: Ke Tang.)*

Shengcai Liu is with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China (e-mail: liuscyyf@mail.ustc.edu.cn).

Ke Tang and Xin Yao are with the Guangdong Provincial Key Laboratory of Brain-Inspired Intelligent Computation, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: tangk3@sustech.edu.cn; xiny@sustech.edu.cn).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCYB.2020.2984546.

Digital Object Identifier 10.1109/TCYB.2020.2984546

[1] According to https://en.wikipedia.org/wiki/FLOPS#Hardware_costs, the unit cost of computing power falls by an order of magnitude roughly every four years nowadays.

for these approaches, an indispensable assumption is that the training set is a good representative of the target use cases [19], such that the "trained" solvers can generalize well to the instances out of the training set. In practice, given a specific application, it could usually be expected that some data, that is, the instances that have been encountered for this application before, are available as the training data. However, it is noted in at least two cases, such a training set might not be sufficiently representative, which could have a major impact on the applicability of the constructed solvers. First, only a limited number of instances are accumulated and thus can hardly cover the entire possible target cases. Second, the accumulated instances are outdated and could not reflect the properties of the current cases well. Actually, the above two cases are not rare and have been discussed in different areas in the literature. For example, it has been reported that in combinatorial optimization, some commonly used benchmark instances are not necessarily challenging [20], narrowly defined [21], and distinct from real-world instances [22]; in research areas closely related to real-world applications, such as logistics, there are also concerns that the instances proposed decades ago already could not represent the real-world cases of today due to the constant growth of big cities [23], [24].

Intuitively, to handle this issue, generating some additional instances appears to be an alternative. However, it is also nontrivial to generate good training data in practice. Recall that the ultimate goal for having a representative training set is to achieve good generalization of the constructed solvers. Thus, the term "representative training set" depends on the specific solvers considered, while the latter is to be constructed based on the former. In other words, it is very difficult to obtain a concrete definition of representativeness in advance, which is crucial for evaluating a given training set and thereby generating a representative one. This difficulty could be alleviated by allowing some redundancy in the training set, since in the extreme case, one could obtain perfect generalization if all possible target instances are included in the training set. However, this could lead to an overwhelming cost when obtaining the training set as well as for constructing the solvers.

This article studies effective construction approaches for parallel portfolios that are less affected by nonrepresentative training data. The term "parallel portfolio" [25], [26] refers to a portfolio/set of solvers that is run independently in parallel when solving a problem instance (see Section III-A). As a form of solvers, parallel portfolios have several important advantages. First, exploiting parallelism has become very important in designing efficient solvers for computationally hard problems, considering the great development and the wide application of parallel computing architectures [27] (e.g., multicore CPUs) over the last decade. Parallel portfolios employ parallel solution strategies and, thus, could easily make effective use of modern hardware. Second, utilizing several different solvers (as in parallel portfolios) is a simple yet effective strategy for solving computationally hard problems. Such an idea has also been realized in the form of sequential portfolios [28], [29], which try to select the best solvers for solving a problem instance, and adaptive solvers, such as adaptive parameter control [30]–[33]; reactive search [34], [35]; and hyper-heuristics [36]–[38], which seek to dynamically determine the best solver setting while solving a problem instance. In principle, all these methods need to involve some mechanisms (e.g., selection or scheduling) to appropriately allocate computational resources to different solvers, while the parallel portfolios do not necessarily require any extra resource allocation since each solver is simply assigned with the same amount of resources. Third, a parallel portfolio could be easily converted to a sequential portfolio by using the algorithm selection methods [39] to build selectors on the solvers in the portfolio, which means the portfolios generated by construction approaches (e.g., the approach proposed in this article) could be further used for constructing sequential portfolios.

In this article, we propose a novel approach called the generative adversarial solver trainer (GAST) for the automatic construction of parallel portfolios. Unlike the existing construction approaches, GAST would generate additional training instances and construct a parallel portfolio with the dynamically changing training set. More specifically, GAST puts instance generation and portfolio construction in an adversarial game. The instance generation aims to generate the hard problem instances that could not be solved well by the current portfolio, while the portfolio construction aims to find a new component solver for the portfolio to better solve these challenging instances. Competition in this game drives the portfolio to satisfactorily solve more problem instances, leading to a better generalization performance. To the best of our knowledge, this is the first work that simultaneously considers solver construction and instance generation. In the experiments, in comparison with the previous approaches, GAST consistently built parallel portfolios with much better generalization across different experimental scenarios, and the portfolios could even achieve the performance level of parallel solvers designed by human experts.

The remainder of this article is organized as follows. Section II reviews previous related work. In Section III, first the problem of parallel portfolio construction is described, and then the general framework of GAST is presented. After that, in Section IV, GAST is further instantiated for TSP and SAT. In Section V, the advantages of GAST will be demonstrated through comparison against other portfolio construction methods in data-scarce and data-biased scenarios. In this section, the portfolios generated by GAST would also be compared against the state-of-the-art manually designed parallel solvers. Finally, the conclusion and future work will be drawn in Section VI.

II. Related Work

A. Automatic Solver Construction

Investigations on automatic solver construction were initiated by the attempts on AAC [9]. A number of algorithm configuraotrs (i.e., AAC methods), ParamILS [6], GGA [10], irace [8], and SMAC [11], have been developed in the past decade. All these methods can be viewed as sharing a common iterative search framework, that is, candidate configurations are generated and tested iteratively. The biggest difference between them lies in the ways of generating candidate configurations. ParamILS and GGA utilize direct search methods, that is, an iterated local search algorithm and a gender-based

genetic algorithm, respectively, to search the configuration spaces, while SMAC and irace both rely on built meta models to guide the sampling of the configuration spaces. With the effective algorithm configurators, there were later attempts to automatically constructing sequential solvers. One prominent example is SATenstein [12], in which ParamILS was used to construct an effective solver for SAT based on a highly parameterized solver framework. Another example is AutoMOEAs [13], in which high-performance multiobjective evolutionary algorithms (MOEAs) for the multiobjective permutation flow-shop problems were built by irace with a configuration space defined on a highly parameterized MOEA framework.

By considering more complicated structures of solvers, research evolved into the realm of automatic portfolio construction (APC), that is, the targeted object is no longer a single solver, but is a portfolio of solvers that are chosen from a configuration space. Such a setting essentially means the search space considered by APC is generally much larger than that considered in the case of constructing sequential solvers, providing more degree of freedom on the resultant solvers and hopefully leading to better performance. According to the ways of using the resultant portfolios to solve a new problem instance, APC was further developed along several directions. Cedalion [16] is a notable approach for constructing portfolios with scheduling for the planning problem, which runs its component planners sequentially with preallocated time budgets. For portfolios with selectors which select a single best solver from its component solvers to solve a given problem instance, there are two representative approaches dubbed Hydra [14] and ISAC [15]. Hydra constructs a portfolio iteratively by finding a configuration in each iteration that maximizes marginal contribution to the current portfolio, while ISAC clusters the training instances based on features and independently runs an algorithm configurator on each cluster. The basic ideas of Hydra and ISAC were later adapted to be used in constructing parallel portfolios, thus resulting in two new approaches PARHYDRA and CLUSTERING [17]. Another key approach for constructing parallel portfolios is PCIT [18], which also adopts an instance grouping strategy such as CLUSTERING but will adjust the grouping by transferring instances between subsets in the construction process. Note that how to evaluate candidate portfolios in the construction process depends on the ways of using the resultant portfolios; therefore the latter should be taken into account in the design of an APC approach.

As mentioned before, currently all investigations on automatic solver construction require that a training set is given, and it is assumed that the training set is a (representative) part of the target use cases. Hence, it is nonsurprising that most of the above approaches were justified on well-investigated computationally hard problems, such as the planning problems [16]; SAT [12], [14], [15], [17], [18]; and TSP [18], since for these problems, there are quite a few benchmark suites. For these approaches, the training set and the test set for empirical studies were usually obtained by randomly and evenly splitting an existing benchmark set into two disjoint sets, such that the training instances can represent the test instances well. However, as aforementioned such a setting could not be always appropriate since in some cases, we might only have scarce and biased training instances.

B. Problem Instance Generation

The lack of instances, though less discussed in the context of automatic solver construction, has attracted much attention from the perspective of empirical evaluation of solvers. In this area, the main goal is to automatically generate problem instances with diverse characteristics, such as hardness and problem features. Various instance generation methods have been proposed to problem domains, such as TSP [40]–[45]; SAT [40], [46]; job-shop scheduling problems [47]; constraint satisfaction problems (CSPs) [40], [48]; graph-coloring problems [21]; and bin-packing problems [49]. The generated instances are usually further used for comprehensive analysis of the strengths and weaknesses of the existing solvers [40]–[43], [45], [48], [49]; algorithm performance prediction [41], [42], [45]; and algorithm enhancement [44], [47].

C. Generative Adversarial Networks

The general idea of GAST is similar to generative adversarial networks (GANs) [50]. GANs also maintain an adversarial game in which a discriminator is trained to distinguish real samples from fake samples synthesized by a generator, and the generator is trained to deceive the discriminator by producing ever more realistic samples. However, there are some main differences between GAST and GANs. First, the overall goals of them are different. GANs focus on the generative models that could capture the distribution of complicated real-world data. For GAST, the main goal is to build powerful parallel portfolios (analogous to the discriminative models in GANs); while the instance generation module as well as the generated instances are more like byproducts. Second, the domains to which GAST and GANs are applicable are different. Currently GANs (and the more general idea of adversarial learning) are mostly successfully applied to vision-related domains, such as image generation [51], [52]; image dehazing [53]; style transfer [54], [55]; image classification [56]; and clustering [57], [58]. In comparison, GAST is proposed for problem-solving domains, such as planning and optimization. Third, the main technical issues in the two areas are different. The ones faced by GANs are the difficulties in modeling complex and large-scale real-world datasets (e.g., mode collapse problem) as well as optimizing the large-scale deep neural networks used in GANs. It has been observed that appropriate hyperparameters are crucial for GANs to work well, and there have been a lot of efforts [59]–[61] dedicated to overcoming these difficulties. For GAST, the main difficulties lie in two aspects: 1) how to generate useful instances for portfolio construction and 2) how to appropriately integrate the instance generation into the portfolio construction process, such that the portfolio's generalization performance would be kept getting improved.

III. Generative Adversarial Solver Training

A. Parallel Portfolios

A parallel portfolio with k component solvers is denoted as a k-tuple $c_{1:k} = (c_1, \ldots, c_k)$, in which c_i represents the ith component solver of $c_{1:k}$. When solving a problem instance, all component solvers of $c_{1:k}$, that is, $c_1, \ldots, c_k$, are run independently in parallel until some termination condition is met. Here, the termination condition may vary according to the problem domains considered and the performance metrics of interest. When a decision problem (e.g., SAT) is considered, all component solvers will be terminated once any of them outputs an answer to the instance, that is, SATISFIABLE or UNSATISFIABLE. In this case, the runtime needed by $c_{1:k}$ to solve the instance is the runtime needed by the best component solver for solving this instance. Moreover, usually a cut-off time, that is, maximum runtime, will be introduced in this case to prevent the solution process from being prohibitively long in which no component solver could solve the problem instance. On the other hand, if an optimization problem (e.g., TSP) is considered, the termination conditions are different according to the performance metrics of interest. If the metric considered is the runtime needed to find a good enough solution of accepted quality level (e.g., within a predefined gap to the optimum), the termination condition is that any of the component solvers finds such a solution. As in the case of the decision problem, a cut-off time could be introduced in this case to prevent the solution process from being prohibitively long. If the metric considered is the quality of the best solution found within a time budget, each component solver will be terminated when the time budget is exhausted and the best solution among the ones found by the component solvers will be returned as the output of $c_{1:k}$.

Overall, the performance of $c_{1:k}$ on an instance s, denoted as $P(c_{1:k}, s)$, is the best performance achieved among $c_1, \ldots, c_k$ on s

$$P(c_{1:k}, s) = \min_{j \in \{1,\ldots,k\}} m\big(c_j, s\big) \tag{1}$$

where $m(c_j, s)$ is the performance of c_j on s according to a performance metric m (e.g., runtime or solution quality). Without loss of generality, we assume a smaller value is better for m. Note that in practice, when an optimization problem is considered, the runtime metric might not be measurable. The reason is that usually, we do not know whether the found solutions by the component solvers are of acceptable quality levels (thus terminating all component solvers), since the optimal solutions of the problem instances are unknown. However, this does not affect the above definition. The performance of $c_{1:k}$ on an instance set I is an aggregated value of the performances of $c_{1:k}$ on all instances in I. Specifically, the following weighted average function, which is widely used in the literature, is used for calculating the performance of $c_{1:k}$ on I, that is, $P(c_{1:k}, I)$:

$$P(c_{1:k}, I) = \frac{1}{|I|} \sum_{s \in I} w_s \cdot P(c_{1:k}, s) \tag{2}$$

where $|I|$ refers to the number of the instances in I and the weight w is introduced to handle different scales of the performances on different instances (usually used when m is related to the solution quality).

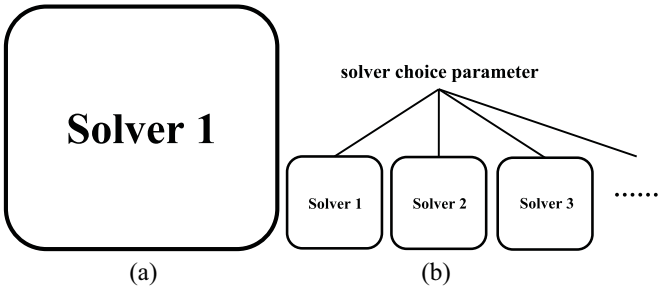


Fig. 1. Configuration space C induced by a set of parameterized solvers B in cases of $|B| = 1$, that is, $B = \{Solver1\}$ in (a) $|B| > 1$, that is, $B = \{Solver1, Solver2, \ldots\}$ in (b), respectively. Each rounded rectangle in this figure represents the parameter space of the corresponding base solver.

B. Problem of Parallel Portfolio Construction

When constructing a portfolio $c_{1:k}$ with AAC, each component solver of $c_{1:k}$ is an individual configuration selected from a configuration space C, that is, $c_1, \ldots, c_k \in C$. C is induced by a set of parameterized solvers B, called the base solvers. As illustrated in Fig. 1, if there is only one base solver, the configuration space is exactly the solver's parameter space; otherwise, the configuration space takes each base solver's parameter space as a subspace, and would include an additional top-level parameter to decide which subspace (base solver) would be used. The full configuration space of $c_{1:k}$ is $C^k = \prod_{i=1}^{k} \{c | c \in C\}$, where the product of two configuration spaces A and B is the Cartesian product of A and B, that is, $A \times B = \{(a, b) | a \in A \text{ and } b \in B\}$. In other words, the size of the full configuration space for $c_{1:k}$ is $|C|^k$.

Given the above definitions, the parallel portfolio construction problem considered here can be stated as follows. Given a possibly nonrepresentative training set I, a performance metric m, a set of parameterized base solvers B, and the configuration space C induced by B, select configurations $c_1, \ldots, c_k$ from C to form a parallel portfolio $c_{1:k}$, such that $c_{1:k}$ can generalize well, that is, achieve good $P(c_{1:k}, I^*)$ on the target set I^*, which is impossible to enumerate in advance, for example, of huge size or is changing over time.

C. Generative Adversarial Solver Trainer

Overall, there are two key design principles for GAST.

The first concerns generating useful training instances. The nonrepresentative training set generally means that some target cases are not covered. It is thus necessary to generate additional training instances. On the other hand, the instances that are out of the training set but can already be solved well by the solvers being constructed are actually of no use for improving the generalization of the solvers. Hence, a desirable generated instance should be not present in the training set and meanwhile hard for the solvers being constructed.

The second principle concerns the complementarity [14], [15], [17], [18], [62] among the component solvers, which is crucial for the effectiveness of any parallel portfolio. According to (1), the performance of a parallel portfolio on

Algorithm 1 GAST

Input: base solvers B with configuration space C; number of component solvers k; instance set I; performance metric m; algorithm configurator AC; independent configurator runs n; time budgets t_C, t_V, t_I for configuration, validation and instance generation respectively
Output: parallel portfolio $c_{1:k}$

1: **for** $i \leftarrow 1 : k$ **do**
2: /*————**configuration phase**————*/
3: **for** $j \leftarrow 1 : n$ **do**
4: obtain a portfolio $c^j_{1:i}$ by running AC on configuration space $\{c_{1:i-1}\} \times \{c | c \in C\}$ using m for time t_C
5: **end for**
6: validate $c^1_{1:i}, ..., c^n_{1:i}$ on I using m for time t_V
7: let $c_{1:i} \leftarrow \arg\min_{c^j_{1:i} | j \in \{1,...,n\}} P(c^j_{1:i}, I)$ be the portfolio with the best validation performance
8: /*————**instance-generation phase**————*/
9: **if** $i = k$ **then break** //skip instance generation
10: according to the validation results, assign the quality score of each $s \in I$ as $w_s \cdot P(c_{1:i}, s)$
11: $\bar{I} \leftarrow I$
12: **while** time spent in this phase not exceeds t_I **do**
13: $I_{new} \leftarrow \varnothing$
14: **for** each $s \in I$ **do**
15: *refset* $\leftarrow$ randomly sample from $I \setminus \{s\}$
16: $s_{new} \leftarrow$ ***variation***$(s, refset)$
17: $I_{new} \leftarrow I_{new} \cup \{s_{new}\}$
18: **end for**
19: test $c_{1:i}$ with each $s \in I_{new}$ and assign the quality score of s as $w_s \cdot P(c_{1:i}, s)$
20: $I \leftarrow I \cup I_{new}$
21: remove $|I_{new}|$ instances from I with binary tournament selection
22: **end while**
23: $I \leftarrow I \cup \bar{I}$
24: **end for**
25: **return** $c_{1:k}$

an instance depends on the best-performing component solver on the instance. Since it is unlikely that a unique component solver performs the best in all instances, it is more desirable that different component solvers are good at solving different problem instances. In other words, GAST should promote different component solvers to handle different instances.

The pseudocode of GAST is given in Algorithm 1. Overall, GAST has an iterative structure and each iteration of GAST consists of two subsequent phases: 1) a configuration phase (lines 3–7) and 2) an instance-generation phase (lines 9–23). The configuration phase is similar to PARHYDRA [17] in which the component solvers of $c_{1:k}$ are configured iteratively. More specifically, in the ith iteration, GAST uses an algorithm configurator (AC in Algorithm 1) with a time budget t_C to configure c_i to add to the current portfolio $c_{1:i-1}$, that is, $(c_1, \ldots, c_{i-1})$, such that the performance of the resulting portfolio $c_{1:i}$, that is, $(c_1, \ldots, c_i)$, on instance set I, is optimized (line 4). During the configuration process of c_i (line 4), GAST would run the entire portfolio on the considered instances while only c_i is available to be configured, leaving $(c_1, \ldots, c_{i-1})$ fixed. In other words, in each iteration, GAST aims to find a configuration that maximizes marginal performance contribution across the configurations identified in the previous iterations. Since generic algorithm configurators are usually randomized methods, to ensure the reliability of the outputs of the algorithm configurator AC, following the established best practices [6], [11], GAST always performs n independent runs of AC when configuring c_i (line 3) and thus obtains n different portfolios produced by these runs, that is, $c^1_{1:i}, \ldots, c^n_{1:i}$. These portfolios are then tested on I with a time budget t_V (line 6) and the one achieving the best validation performance will be retained (line 7).

The instance-generation phase begins once the configuration phase finishes. Note in the last iteration (i.e., the kth iteration) of GAST, instance generation is skipped (line 9) because there is no need to generate more instances since $c_{1:k}$ has been completely constructed. In the instance-generation phase, GAST first creates a backup of the training set I (line 11) that will be restored to the training set at the end of this phase (line 23), and then enters an iterative process in which GAST repeatedly generates new instances based on the current training set I (lines 12–18), tests these new instances with current portfolio $c_{1:i}$ (line 19) and uses them to update the instance set I (lines 20 and 21), until the time spent for generating instances reaches budget t_I (line 12).

More specifically, to generate a new instance s_{new}, GAST uses an existing instance s in I as a base instance, and randomly selects a set of instances from I excluding s as the reference instances (*refset* in line 15). s_{new} is then generated by modifying s with random perturbation and insertion of structures/components extracted from the reference instances (by the *variation* procedure in line 16). Taking each instance in I as the base instance (line 14), GAST eventually generates a set of new instances I_{new}. The instances generated in this way are expected to differ significantly from the existing instances in I, but at the same time, would preserve some characteristics of the existing ones. This is desirable because generating too similar instances to the existing ones is not useful for exploring the instance space, which is crucial for improving the generalization of the portfolio being constructed, and generating instances completely unrelated to existing ones could result in instances of no interests, for example, instances with no practical significance. Moreover, since each existing instance in I is used as the base instance to generate new ones, the diversity in I_{new} is expected to be enhanced. The precise definition of the modification procedure depends on the specific problem domain considered; thus it is encapsulated as the *variation* procedure in Algorithm 1 (line 16). A lot of existing instance variation mechanisms, which are applicable for a wide range of problem domains (see Section II), could be used to instantiate *variation* when applying GAST to the corresponding domains (see Section IV for the instantiations for TSP and SAT).

Another two important aspects in the instance-generation phase are the instance evaluation and the instance selection. As aforementioned, only instances that cannot be solved well by the current portfolio $c_{1:i}$ is valuable for improving the generalization of the portfolio; thus in the instance-generation phase, each instance is assigned with a quality score equal to the performance of the current portfolio on it [i.e., $w_s \cdot P(c_{1:i}, s)$]—the worse the performance, the higher the score (note w_s is just the normalization factor to handle different scales

of the performances). For the initial instances in I when entering instance-generation phase, their quality scores could be directly obtained from the validation results which were cached in the configuration phase (line 10). As for those newly generated instances, GAST will test them with $c_{1:i}$ (line 19) and obtain their quality scores.[2] After that, all newly generated instances, that is, instances in I_{new}, are included in the training set I (line 20), and then binary tournament selection [63], which repeatedly randomly selects two instances from I and removes the one with the lower quality score, is used to remove $|I_{\text{new}}|$ instances from I to keep its size unchanged.

In general, GAST alternates between generating new training instances that are hard for the current portfolio and configuring a new component solver to solve these instances while leaving the existing component solvers clamped. In this sense, GAST always promotes the component solver being configured in the current iteration to handle the newly generated instances which are different from the ones considered in the previous iterations, such that the complementarity among the component solvers of the constructed portfolio would be enhanced.

D. Discussion

Intuitively, if we consider an instance "covered" by a portfolio $c_{1:k}$ as it can be solved well by $c_{1:k}$; then the target of the construction problem considered here is to find $c_{1:k} = (c_1, \ldots, c_k)$ from the configuration space C with a maximum coverage on the target instance space I^*. Generally, the problem is NP-hard and can be approximated within $1-(1/e)+o(1) \approx 0.632$. The approximation ratio is achieved by the generic greedy method [64]. More specifically, this is an iterative method which starts from an empty portfolio and at each iteration selects a configuration from C that covers the largest number of uncovered instances in I^* to add to the portfolio. The iterative framework of GAST (the outermost loop in Algorithm 1) is exactly the same as the greedy method except that GAST involves an additional instance-generation phase in each iteration. Recall that in the problem considered here, we are only given a training set I that is nonrepresentative of I^*, and I^* is impossible to enumerate in advance, for example, of huge size or is changing over time. This means during the portfolio construction it is unclear which instances in I^* are not covered by the current portfolio. Thus, it is necessary to first identify those uncovered instances in I^* for enlarging the portfolio's coverage on I^*, which is exactly what the adversarial instance generation does. In comparison, the existing approaches do not involve such mechanisms; thus they could only optimize the portfolio's coverage on the training set I. For the instances that are in I^* but not in I, the portfolio's performance is not optimized and could be arbitrarily bad.

E. Time Complexity and Computational Costs

The most time-consuming parts of GAST are the runs of the component solvers on the problem instances, and the incurred computational costs account for the vast majority of the total costs of GAST. Therefore, we analyze the time complexity of GAST in terms of the total number of the runs of the solvers. In Algorithm 1, the solvers are invoked in three places, that is, configuration (line 4), validation (line 9), and instance generation (line 19). Recall that in the ith iteration of GAST, there are i component solvers in the portfolio and they are always executed in parallel. Let N_C, N_V, and N_I denote the number of the runs of each component solver in configuration (line 4), validation (line 9), and instance generation (line 19), respectively. The total number of runs of solvers in the ith iteration of GAST is $i \cdot [n \cdot (N_C + N_V) + N_I]$, where n is the number of independent configurator runs (line 3). Considering the instance-generation phase is skipped in the last iteration of GAST, the time complexity of GAST in terms of the number of the runs of the solvers is $O(\sum_{i=1}^{k} i{\cdot}n{\cdot}(N_C{+}N_V){+}\sum_{i=1}^{k-1} i{\cdot}N_I) = O(k^2 n(N_C + N_V) + k^2 N_I)$. Similarly, we could obtain that the time complexity of the existing parallel portfolio construction approaches, that is, PARHYDRA, GLOBAL, and PCIT, are $O(k^2 n(N_C + N_V)$, $O(kn(N_C + N_V))$, and $O(kn(N_C + N_V))$, respectively. For detailed information of how these results are derived, we refer the reader to the original articles [17] for PARHYDRA and GLOBAL and [18] for PCIT. Note that for a specific portfolio construction approach, the values of N_C, N_V, and N_I depend on the predefined time budgets t_C, t_V, and t_I, respectively, and for different approaches, t_C, t_V, and t_I could be set differently.

TABLE I
COMPUTATIONAL COSTS FOR EXISTING PARALLEL PORTFOLIO CONSTRUCTION APPROACHES. t_C, t_V, AND t_I ARE TIME BUDGETS FOR CONFIGURATION, VALIDATION, AND INSTANCE GENERATION, RESPECTIVELY. k IS THE PORTFOLIO SIZE. n IS THE NUMBER OF INDEPENDENT RUNS OF THE USED ALGORITHM CONFIGURATOR. NOTE THAT FOR DIFFERENT APPROACHES t_C, t_V COULD BE SET TO DIFFERENT VALUES

	CPU time
GAST	$\sum_{i=1}^{k} i \cdot n \cdot (t_C + t_V) + \sum_{i=1}^{k-1} i \cdot t_I$
PARHYDRA	$\sum_{i=1}^{k} i \cdot n \cdot (t_C + t_V)$
GLOBAL	$n \cdot k \cdot (t_C + t_V)$
PCIT	$n \cdot k \cdot (t_C + t_V)$

Given the time budgets t_C, t_V, and t_I, the total CPU time consumed by GAST is $\sum_{i=1}^{k} i \cdot n \cdot (t_C + t_V) + \sum_{i=1}^{k-1} i \cdot t_I$. The n independent runs of AC (line 4) and the validation processes (line 6) can be performed in parallel if n machines (with each of k cores) are available, in which case GAST will require $k \cdot (t_C + t_V) + (k-1) \cdot t_I$ wall clock time to complete. For completeness, in Table I, we also list the needed CPU time for PARHYDRA, GLOBAL, and PCIT, which will be referenced in the experiments (see Section V-A4).

IV. INSTANTIATIONS OF GAST FOR TSP AND SAT

In this section, the *variation* procedure in GAST is instantiated for TSP and SAT, respectively, resulting in two approaches GAST-TSP and GAST-SAT.

A. GAST-TSP

Specifically, the symmetric TSP, that is, the distance between two cities is the same in each opposite direction,

[2]When testing the generated instances with $c_{1:i}$, if the component solvers in $c_{1:i}$ are randomized solvers, which is actually very common, $c_{1:i}$ will be run on each instance for several times with different random seeds, and the mean value of the test results will be used.

with distances in a 2-D Euclidean space is considered here. Each instance of such TSP is represented by a list of (x, y) coordinates with each coordinate as a city. We extended the variation strategy used in [40] which requires the base instances and the reference instances have the same size (i.e., number of the cities) to allow the use of instances of different sizes. Specifically, given a base instance s, and a reference instance s^* (meaning GAST-TSP requires only one reference instance in *refset*; see lines 15 and 16 in Algorithm 1), the *variation* procedure in GAST-TSP applies a variable-length crossover and a uniform mutation to s and s^* to generate a new instance. Let $|s|$ and $|s^*|$ be the length of the coordinate list of s and s^*, respectively. The crossover first randomly selects $\min\{|s|, |s^*|\}-1$ split points in both lists, and then constructs a new coordinate list (i.e., the new instance s_{new}) in a sequential manner by choosing each segment from either of the two lists with equal probability. The new list is then subject to the mutation operator that replaces each coordinate in the list, with a probability $1/|s_{\text{new}}|^{(1/2)}$, with a coordinate uniform randomly chosen within the ranges bounded by the minimum and the maximum values of the coordinates in the lists of s and s^*.

B. GAST-SAT

The *variation* procedure in GAST-SAT utilizes the *spig* technique proposed by [46], which iteratively removes particular components (bounded together through a core variable) from the base instance s and then inserts such structures extracted from the reference instances into s. The generated instance will only be accepted by *spig* if all of its features are within σ standard deviations of the mean value across all instances (including s and the reference instances). The value of σ is set to a quite small value, that is, 3, by [46] for generating similar enough instances to the existing ones, which obviously is not our goal here. We thus set σ as a random variable whose value is randomly sampled from $[3, 300]$ for each acceptance check to introduce more randomness in the generated instances. To prevent the runtime of *spig* from being too long, the size of the reference instance set, that is, $|refset|$, is set to $\lceil |I|^{(1/2)} \rceil$, where $|I|$ is the size of the training set.

V. Experiments

We conducted experiments on SAT and TSP. Following the common scheme, in the experiments, we used GAST to build parallel portfolios based on a training set, and then compared them against the ones constructed by the existing approaches, on an unseen test set.

A. Experimental Setup

1) Portfolio Size and Performance Metric: We set the number of component solvers k to 4, since 4-core machines are widely available now. The optimization goal considered here is the runtime needed by a solver to solve the problem instances (for SAT) or to find the optimums of the problem instances (for TSP). In particular, we set m to *Penalized Average Runtime*-10 (PAR-10) [6], which is the average runtime over all the test runs, where those unsuccessful runs (unable to solve the given instance within the cut-off time) are counted as ten times the cut-off time. Note for PAR-10, the weight w in (1) is set to 1. The optimal solutions for TSP instances were obtained using Concorde [67], an exact TSP solver.

2) Instance Sets: Since we focus on the scenarios where the available training instances are nonrepresentative, it is very important to decide an appropriate way to choose the instances. We used two different ways to obtain the instances, thus dividing our experiments into two parts. In the first part, we obtained instances through instance generators and evaluated GAST-TSP in this part because for TSP there exist generators that could generate instances with diverse characteristics. Specifically, we used the *portgen* and the *portcgen* generators from the eighth DIMACS implementation challenge [68] to generate 150 "uniform" instances (in which the cities are randomly distributed) and 150 "clustering" instances (in which the cities are distributed around different central points) to form a set of 300 instances, denoted as $\text{TSP}_{\text{whole}}$. The problem sizes of all these generated instances are within $[400, 600]$.

The instance-generation way has two potential issues. First, the generated instances might be far away from the real-world cases, thus making the evaluation on them not of practical significance. Second, since GAST also involves instance generation (see Algorithm 1), there is a possibility that the underlying generation model in GAST is similar to the instance generators used here. To avoid these issues, in the second part, we only obtained the instances from the industrial benchmark suites and evaluated GAST-SAT in this part. Specifically, we obtained two industrial benchmarks, IBM hardware verification (HV) benchmark and bounded model checking (BMC) benchmark, from the algorithm configuration library (AClib) [7], and randomly selected 150 instances from each of the two sets to form a set of 300 instances, denoted as $\text{SAT}_{\text{whole}}$.

3) Experimental Scenarios: For brevity, we only describe how we split $\text{TSP}_{\text{whole}}$ here. For $\text{SAT}_{\text{whole}}$, the same procedure was conducted. We split $\text{TSP}_{\text{whole}}$ into training sets and test sets in two different ways, for simulating two different cases. The first case "SMALL" means the available training set contains only a small number of instances. In this case, we randomly selected 1/6 instances (50 in total) from $\text{TSP}_{\text{whole}}$ as training instances and used the left instances (250 in total) as test instances. The second case "BIAS" means the training instances are biased to narrowly defined cases. In this case, from $\text{TSP}_{\text{whole}}$, we randomly selected 1/3 instances from one of the two types of the instances (50 in total; recall that there are 150 uniform instances and 150 clustering instances in $\text{TSP}_{\text{whole}}$) as training instances, and used the left instances (250 in total) as test instances.

Since the above split procedure is randomized and the choices of training/test instances would obviously affect the performances of portfolio construction approaches, to ensure the reliability of our experiments, we repeated the above split procedure four times for each of SMALL and BIAS cases, which eventually gave us eight different experimental scenarios, with each of a unique pair of training set and test set, for each of the TSP and SAT domains. For convenience, we use TSP-SMALL/BIAS-1/2/3/4 and SAT-SMALL/BIAS-1/2/3/4 to denote these scenarios. Moreover,

TABLE II
SUMMARY OF THE INSTANCE SETS, THE CUT-OFF TIME, AND THE BASE SOLVER IN EACH SCENARIO

	Instance Sets	Cut-off Time	Base Solvers
TSP-SMALL	Randomly select 50 instances from $\text{TSP}_{\text{whole}}$ as training instances and the rest 250 instances are used as test instances	1s	LKH version 2.0.7 [65] with 23 parameters
TSP-BIAS	Randomly select 50 instances from 150 "uniform" instances or 150 "clustering" instances in $\text{TSP}_{\text{whole}}$ as training instances and the rest 250 instances are used as test instances		
SAT-SMALL	Randomly select 50 instances from $\text{SAT}_{\text{whole}}$ as training instances and the rest 250 instances are used as test instances	150s	lingeling-ala [66] with 118 parameters
SAT-BIAS	Randomly select 50 instances from 150 "IBM HV" instances or 150 "BMC" instances in $\text{SAT}_{\text{whole}}$ as training instances and the rest 250 instances are used as test instances		

TABLE III
DETAILED TIME BUDGET IN TERMS OF HOURS OF CPU TIME FOR EACH APPROACH IN EACH SCENARIO. "TSP" REPRESENTS EIGHT SCENARIOS TSP-SMALL/BIAS-1/2/3/4 AND "SAT" REPRESENTS EIGHT SCENARIOS SAT-SMALL/BIAS-1/2/3/4. IN THE EXPERIMENTS, THE NUMBER OF INDEPENDENT RUNS OF ALGORITHM CONFIGURATOR n FOR ALL APPROACHES WERE SET TO 10. t_C, t_V, AND t_I ARE TIME BUDGETS FOR CONFIGURATION, VALIDATION, AND INSTANCE GENERATION, RESPECTIVELY. SEE TABLE I FOR HOW TO ESTIMATE THE NEEDED CPU TIME FOR EACH APPROACH

	TSP				SAT			
	t_C	t_V	t_I	total	t_C	t_V	t_I	total
GAST	1.5h	0.5h	5h	230h	8h	4h	40h	1440h
GLOBAL	5h	0.5h	–	220h	30h	4h	–	1360h
PCIT	5h	0.5h	–	220h	30h	4h	–	1360h
PARHYDRA	1.5h	0.5h	–	230h	8h	4h	–	1200h

we use TSP-SMALL to denote a set of four scenarios {TSP-SMALL-1/2/3/4}, and the same rule applies to TSP-BIAS, SAT-SMALL, and SAT-BIAS.

Table II summarizes the instance sets, the cut-off time, and the base solvers used in different scenarios. The base solver used in TSP-SMALL/BIAS was LKH version 2.0.7 [65] (with 23 parameters), one of the state-of-the-art inexact solver for TSP. The base solver used in SAT-SMALL/BIAS was lingeling-ala [66] (with 118 parameters), the gold medal-winning solver in the application track of the 2011 SAT Competition.

4) Competitors and Time Budgets: We compared GAST against the state-of-the-art automatic construction approaches for parallel portfolios: GLOBAL, PARHYDRA [17], and PCIT [18]. For all considered approaches here, SMAC version 2.10.07 [11] was used as the algorithm configurator. Since the performance of SMAC could be often improved when used with the instance features, we gave SMAC access to the 126 SAT features and the 114 TSP features used in [18]. The detailed setting of the time budget for each approach is given in Table III. Overall, in the experiments, GAST would consume around 10% more CPU time than other approaches for the construction of parallel portfolios.

All the experiments were conducted on a cluster of three Intel Xeon machines with 128-GB RAM and 24 cores each (2.20 GHz, 30-MB Cache), running Centos 7.5. The entire experiments took almost 2.5 months to complete.

B. Results and Analysis

In each experimental scenario, we tested each obtained portfolio by running it on the test set for 50 times (for TSP) and for 5 times (for SAT). The mean $\pm$ stddev of the test performances (PAR-10 score over the test instances) across these runs, and the total number of timeouts (#TOs), are presented in the PAR-$10^{\dagger}$ columns and the #TOs columns, respectively, in Table IV. The validation performances of the portfolios constructed by PARHYDRA, GLOBAL, and PCIT (except for GAST since it keeps changing the training set) on the training sets are also reported in the PAR-$10^{\star}$ columns in Table IV. To determine whether the differences between the test performances (i.e., the PAR-$10^{\dagger}$ columns) were significant, we performed a Wilcoxon signed-rank test (with significance level $p = 0.05$) to them in each scenario, and a PAR-10 score is indicated in boldface if it was not significantly different from the best test PAR-10 score of the scenario.

Overall, GAST is the best-performing approach in Table IV and in most cases, it constructed significantly and substantially better portfolios than the other approaches. Since PARHYDRA could be seen as a variant of GAST without the instance generation mechanism (see Section III-C), the superior performances of GAST over PARHYDRA indicate the effectiveness of the instance generation for improving the portfolio's generalization. Moreover, recall that we used generated instances for TSP and industrial instances for SAT, the consistently strong performances of GAST on both domains indicate that generating new instances through recombination of the existing instances and random perturbation (as in GAST) is a robust and effective way for training data augmentation. Another important observation from Table IV is that for the existing approaches, the gaps between the validation performances and the test performances are usually very large; this is conceivable since in the experiments, the training set is expected to be nonrepresentative, which in turn indicates the necessity of instance generation in this case.

C. Comparison Against PARHYDRA When k Is Larger Than 4

Both GAST and PARHYDRA are iterative approaches, adding one component solver to the portfolio per iteration. To investigate how they would perform when the number of iterations (i.e., portfolio size) gets larger, we run GAST and PARHYDRA for eight iterations ($k = 8$) in four scenarios TSP/SAT-SMALL/BIAS-1. Let GAST_i and PARHYDRA_i denote the resultant portfolios at the end of the ith iteration of GAST and PARHYDRA, respectively. In each scenario,

TABLE IV
RESULTS OF VALIDATION AND TESTING IN THE 16 EXPERIMENTAL SCENARIOS. VALIDATION PERFORMANCES IN TERMS OF PAR-10 SCORES OVER THE TRAINING SET ARE PRESENTED IN THE PAR-10* COLUMNS. TEST PERFORMANCES IN TERMS OF MEAN $\pm$ STDDEV OF THE PAR-10 SCORES ACROSS THE 50 RUNS (FOR TSP) AND THE 5 RUNS (FOR SAT) OVER THE TEST SET ARE PRESENTED IN THE PAR-10† COLUMNS. THE TOTAL NUMBER OF TIMEOUTS (#TOS) IN TESTING IS PRESENTED IN THE #TOS COLUMNS. THE NAME OF THE CONSTRUCTION APPROACH IS USED TO DENOTE THE PORTFOLIOS CONSTRUCTED BY IT. THE TEST PAR-10 SCORE OF A PORTFOLIO IS SHOWN IN BOLDFACE IF IT WAS NOT SIGNIFICANTLY DIFFERENT FROM THE BEST TEST PERFORMANCE IN THE SCENARIO (ACCORDING TO A WILCOXON SIGNED-RANK TEST WITH $p = 0.05$)

	GAST		GLOBAL			PARHYDRA			PCIT		
	PAR-10†	#TOs	PAR-10*	PAR-10†	#TOs	PAR-10*	PAR-10†	#TOs	PAR-10*	PAR-10†	#TOs
TSP-SMALL-1/2/3/4	**0.48 ± 0.09**	327	0.23	1.02 ± 0.24	934	0.18	**0.67 ± 0.18**	565	0.18	0.58 ± 0.12	388
	0.48 ± 0.08	282	0.24	0.71 ± 0.16	521	0.16	0.50 ± 0.13	354	0.19	**0.56 ± 0.12**	390
	0.47 ± 0.11	277	0.29	0.98 ± 0.22	868	0.18	0.50 ± 0.12	356	0.19	**0.48 ± 0.09**	292
	0.51 ± 0.12	319	0.28	1.01 ± 0.29	918	0.16	0.51 ± 0.15	346	0.18	**0.50 ± 0.11**	326
TSP-BIAS-1/2/3/4	**0.68 ± 0.18**	550	0.17	1.04 ± 0.20	925	0.15	0.82 ± 0.18	715	0.18	1.26 ± 0.21	1196
	0.50 ± 0.14	340	0.20	0.72 ± 0.18	590	0.14	0.59 ± 0.16	471	0.20	0.55 ± 0.14	369
	0.62 ± 0.14	434	0.26	0.90 ± 0.20	767	0.13	**0.86 ± 0.20**	753	0.17	0.75 ± 0.16	599
	0.51 ± 0.13	307	0.23	0.76 ± 0.18	604	0.18	**0.60 ± 0.18**	442	0.18	0.50 ± 0.13	333
SAT-SMALL-1/2/3/4	**317 ± 4.93**	258	280	351 ± 15.3	285	255	333 ± 5.66	271	251	327 ± 5.66	266
	283 ± 4.16	230	403	317 ± 7.66	258	398	321 ± 9.39	261	402	322 ± 15.0	262
	335 ± 8.77	274	324	355 ± 9.88	290	307	**334 ± 9.58**	272	310	**374 ± 2.49**	306
	295 ± 4.07	240	279	346 ± 9.65	281	285	336 ± 8.21	273	276	356 ± 10.4	291
SAT-BIAS-1/2/3/4	**299 ± 4.79**	243	377	383 ± 17.9	313	337	353 ± 3.25	288	366	369 ± 16.7	302
	272 ± 6.53	220	517	350 ± 7.42	285	504	299 ± 11.0	241	512	287 ± 11.1	233
	331 ± 4.36	271	250	394 ± 11.2	322	245	355 ± 6.77	290	248	387 ± 8.19	318
	324 ± 5.58	264	393	348 ± 10.2	284	388	**310 ± 7.48**	250	364	334 ± 12.1	273

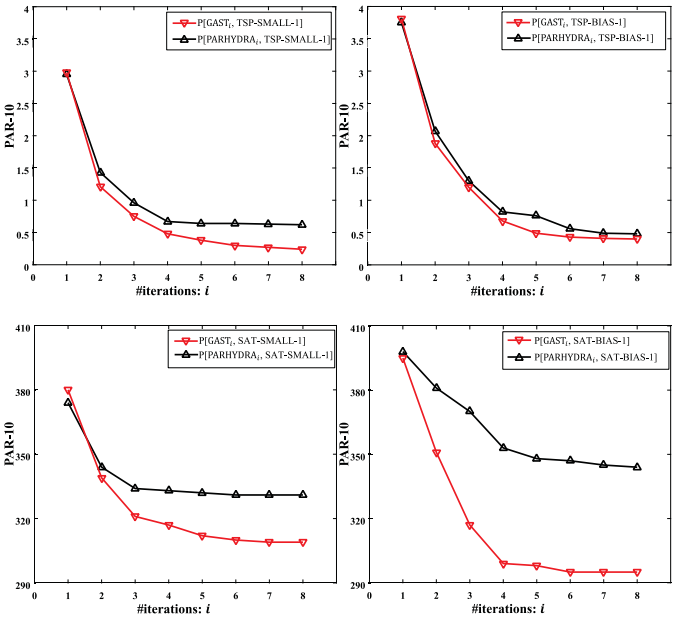


Fig. 2. Test performance (in terms of average PAR-10 scores) progress when $k = 8$ for GAST and PARHYDRA in four scenarios (TSP/SAT-SMALL/BIAS-1) along the number of iterations. GAST$_i$ and PARHYDRA$_i$ are the resultant portfolios at the end of the ith iteration of GAST and PARHYDRA, respectively. P[*portfolio*, *scenario*] is the average test result in terms of PAR-10 scores on the test set in the scenario.

we tested the corresponding GAST$_i$ and PARHYDRA$_i$ with $i = 1, \ldots, 8$ on the test instances, and let *P*[*portfolio*, *scenario*] be the average test result in terms of PAR-10 scores. For example, P[GAST$_2$, TSP-SMALL-1] is the average PAR-10 score of the test result of GAST$_2$, that is, the resultant portfolio at the end of the second iteration of GAST in TSP-SMALL-1, on the test instances of TSP-SMALL-1. The results are plotted along with the number of iterations in Fig. 2. There are three observations from Fig. 2. First, for both GAST and PARHYDRA, the test performance improves monotonically from one iteration to the next. This is reasonable because adding a component solver to an existing portfolio would result in a new portfolio that is theoretically not worse (mostly better) than the original portfolio. Second, for both GAST and PARHYDRA, as the number of iterations increases, the benefits of adding new component solvers gradually decrease. Especially, when the number of iterations becomes larger than 5, the performance improvement is very small. This is conceivable because the performance of the portfolio becomes better and better as the number of iterations increases, which in turn makes it more difficult to further improve the performance of the portfolio. Third, GAST could usually achieve larger performance improvements than PARHYDRA. For example, in SAT-BIAS-1, in the earlier iterations, GAST achieved remarkable performance improvements in comparison with PARHYDRA. This clearly shows the performance of PARHYDRA is limited by the nonrepresentative training data while GAST could break such limitations with the instance generation mechanisms.

D. Comparison Against PARHYDRA With Augmented Training Sets

Since GAST generates instances for configuring the portfolios while the existing approaches do not involve any instance generation, in this sense, GAST actually uses much more instances than the other approaches for construction. A natural question is that, if given enough generated instances, how will the existing approaches perform when compared against GAST? If the former could reach (or even exceed) the performance level of GAST, it could be concluded that it is unnecessary to realize instance generation and portfolio construction simultaneously in an adversarial framework (as in GAST); instead, directly generating enough instances and then using the existing portfolio construction approaches to build portfolios on them is already good enough for handling data-scarce/biased scenarios.

TABLE V
TEST PERFORMANCES IN TERMS OF MEAN ± STDDEV OF THE PAR-10 SCORES ACROSS THE FIVE RUNS OVER THE TEST SET IN THE 8 SAT SCENARIOS, THAT IS, SAT-SMALL-BIAS-1/2/3/4. THE NAME OF THE CONSTRUCTION APPROACH IS USED TO DENOTE THE PORTFOLIOS CONSTRUCTED BY IT. "PARHYDRA-A" REFERS TO PARHYDRA CONFIGURING BASED ON AUGMENTED TRAINING SETS. A PAR-10 SCORE IS SHOWN IN BOLDFACE IF IT WAS NOT SIGNIFICANTLY DIFFERENT FROM THE BEST TEST PERFORMANCE IN THE SCENARIO (ACCORDING TO A WILCOXON SIGNED-RANK TEST WITH $p = 0.05$)

	GAST	PARHYDRA	PARHYDRA-A
SAT-SMALL-1/2/3/4	**317 ± 4.93**	333 ± 5.66	**325 ± 4.89**
	283 ± 4.16	321 ± 9.39	300 ± 8.90
	335 ± 8.77	**334 ± 9.58**	349 ± 6.76
	295 ± 4.07	336 ± 8.21	329 ± 10.4
SAT-BIAS-1/2/3/4	**299 ± 4.79**	353 ± 3.25	331 ± 13.4
	272 ± 6.53	299 ± 11.0	323 ± 12.5
	331 ± 4.36	355 ± 6.77	345 ± 12.8
	324 ± 5.58	**310 ± 7.48**	333 ± 12.4

To answer this question, in each of the eight SAT scenarios, that is, SAT-SMALL/BIAS-1/2/3/4, we used the same instance generation procedure as in GAST (lines 13–18 in Algorithm 1) to generate a large set of instances based on the training set. The size of the generated set is five times the size of the training set. Recall that the training set contains 50 instances, we thus obtained an augmented training set of 300 instances in each SAT scenario, and then PARHYDRA was used to construct a parallel portfolio on these augmented training sets, and then the obtained portfolio was tested on the test sets. As before, each portfolio was tested by running it on the test set for five times. The mean ± stddev of the test PAR-10 scores across the five runs are presented in the "PARHYDRA-A" column in Table V.

For the sake of comparison, the test performances of the portfolios constructed by GAST and PARHYDRA (without augmented training sets) in SAT-SMALL/BIAS-1/2/3/4, which are originally presented in Table IV, are also presented in Table V. It could be seen from Table V that even with augmented training sets, PARHYDRA still could not reach the performance levels of GAST. Note in SAT-SMALL-3 and SAT-BIAS-2/4, when using generated instances, the performance of PARHYDRA would even deteriorate. The key for training set augmentation is which kinds of generated instances should be used. GAST generates instances in an adversarial process where only the hard instances for the current portfolio are selected because on them there is a high opportunity for improvement. This could be seen as a guided sampling in the instance space, which always seeks to find areas not covered by the portfolio yet. On the other hand, treating data augmentation and portfolio construction as two sequential and independent phases, that is, generating enough training instances and then using PARHYDRA to build portfolios on them, lacks such guidance and might cause useless (e.g., too easy or duplicate) instances in the training set, which might be harmful for the portfolio construction (as in the cases of SAT-SMALL-3 and SAT-BIAS-2/4). Overall, GAST is more effective at data augmentation and thus performs better.

E. Comparison Against Hand-Designed Parallel Solvers

To further evaluate the portfolios constructed by GAST, we compared them against the state-of-the-art manually designed parallel solvers. Specifically, we considered the ones constructed for SAT. We tested each of the eight portfolios constructed by GAST in SAT-SMALL/BIAS-1/2/3/4 on the entire SAT instance set, that is, $\text{SAT}_{\text{whole}}$, and reported the best, the worst, and the median performance (in terms of PAR-10) achieved among these portfolios in Table VI. For manually designed solvers, we chose Plingeling-ala [66], which is the official parallel version of lingeling-ala (the base solver in all the SAT scenarios in our experiments), pfolioUZK [69], the gold medal winning solver of the parallel track of the SAT'12 Challenge, and Plingeling-bbc [70], the gold medal-winning solver of the parallel track of the SAT'16 Competition. Note that all the manually designed solvers considered here have implemented far more advanced parallel solving strategies (e.g., clause sharing) than only independently running component solvers in parallel. The default settings of these solvers were used and all of them were tested on $\text{SAT}_{\text{whole}}$. The test performances are presented in Table VI. As before, we performed a Wilcoxon signed-rank test (with significance level $p = 0.05$) to the test performances, and a PAR-10 score is indicated in boldface if it was not significantly different from the best test performance.

TABLE VI
TEST PERFORMANCES IN TERMS OF MEAN ± STDDEV OF THE PAR-10 SCORES ACROSS THE FIVE RUNS OVER $\text{SAT}_{\text{WHOLE}}$. "UZK" REFERS TO PFOLIOUZK. "ALA" REFERS TO PLINGELING-ALA. "BBC" REFERS TO PLINGELING-BBC. "GAST-B," "GAST-W," AND "GAST-M" REFER TO THE BEST, THE WORST, AND THE MEDIAN PERFORMANCE ACHIEVED AMONG THE EIGHT PORTFOLIOS CONSTRUCTED BY GAST IN SAT-SMALL/BIAS-1/2/3/4. A PAR-10 SCORE IS SHOWN IN BOLDFACE IF IT WAS NOT SIGNIFICANTLY DIFFERENT FROM THE BEST TEST PERFORMANCE (ACCORDING TO A WILCOXON SIGNED-RANK TEST WITH $p = 0.05$)

	UZK	ALA	BBC	GAST-B	GAST-W	GAST-M
$\text{SAT}_{\text{whole}}$	317±7.48	310±9.24	**274±4.03**	**282±5.18**	313±5.79	301±8.21

As shown in Table VI, the portfolios constructed by GAST always perform better than pfolioUZK and in most cases perform better than Plingeling-ala. It is impressive to see that in the best case, the portfolio constructed by GAST (regardless of its simple parallel-solving strategy) could reach the performance level of the more state-of-the-art Plingeling-bbc, and moreover the performance difference between them is statistically insignificant. Such results indicate that GAST could identify powerful parallel portfolios, with little human effort involved. It is expected that given more state-of-the-art base solvers, for example, lingeling-bbc, GAST could deliver parallel portfolios with even better performance.

VI. CONCLUSION AND DIRECTIONS FOR FURTHER RESEARCH

This article proposed a novel approach, dubbed GAST, for the automatic construction of parallel portfolios in environments where the training sets are nonrepresentative. The most novel feature of GAST is that, different from existing approaches, it considers instance generation and portfolio construction simultaneously in an adversarial process. Instantiations of GAST for TSP and SAT were also proposed. The experimental results showed that GAST could identify parallel portfolios with much better generalization than

the ones generated by the existing approaches when the training data were scarce and biased. Moreover, it was further demonstrated that the generated portfolios could reach the performance level of the state-of-the-art parallel solvers designed by human experts. Further directions for investigations might include the following:

1) Further improvements to GAST. The diversity preservation scheme, such as speciation [71] or negatively correlated search [72] can be introduced into GAST to explicitly promote cooperation between different component solvers.
2) Deeper understanding of the foundations of GAST. For example, GAST actually maintains two adversary sets competing against one another, which is a typical scenario where the game theory can be applied.
3) Other more general issues in training instance augmentation, for example, a similarity measure between problem instances and instance space characterization, are also worthy of exploration.

REFERENCES

[1] C.-C. Chang and H.-T. Huang, "Automatic tuning of the RBF kernel parameter for batch-mode active learning algorithms: A scalable framework," *IEEE Trans. Cybern.*, vol. 49, no. 12, pp. 4460–4472, Dec. 2019.

[2] Z.-Z. Liu, Y. Wang, S. Yang, and K. Tang, "An adaptive framework to tune the coordinate systems in nature-inspired optimization algorithms," *IEEE Trans. Cybern.*, vol. 49, no. 4, pp. 1403–1416, Apr. 2019.

[3] A. Mohammadi, H. Asadi, S. M. K. Mohamed, K. Nelson, and S. Nahavandi, "Multiobjective and interactive genetic algorithms for weight tuning of a model predictive control-based motion Cueing algorithm," *IEEE Trans. Cybern.*, vol. 49, no. 9, pp. 3471–3481, Sep. 2019.

[4] R. Martinez-Cantin, "Funneled Bayesian optimization for design, tuning and control of autonomous systems," *IEEE Trans. Cybern.*, vol. 49, no. 4, pp. 1489–1500, Apr. 2019.

[5] S. Liu, K. Tang, Y. Lei, and X. Yao, "On performance estimation in automatic algorithm configuration," 2019. [Online]. Available: arXiv:1911.08200.

[6] 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.

[7] F. Hutter *et al.*, "AClib: A benchmark library for algorithm configuration," in *Proc. 8th Int. Conf. Learn. Intell. Optim. (LION)*, Feb. 2014, pp. 36–40.

[8] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari, "The irace package, iterated race for automatic algorithm configuration," IRIDIA, Université Libre de Bruxelles, Belgium, Tech. Rep. TR/IRIDIA/2011-004, 2011. [Online]. Available: http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2011-004.pdf

[9] H. H. Hoos, "Automated algorithm configuration and parameter tuning," in *Autonomous Search*, Y. Hamadi, E. Monfroy, and F. Saubion, Eds. Heidelberg, Germany: Springer, 2012, pp. 37–71.

[10] C. Ansótegui, M. Sellmann, and K. Tierney, "A gender-based genetic algorithm for the automatic configuration of algorithms," in *Proc. 15th Int. Conf. Principles Practice Constraint Program. (CP)*, Sep. 2009, pp. 142–157.

[11] 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)*, Jan. 2011, pp. 507–523.

[12] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, "SATenstein: Automatically building local search SAT solvers from components," in *Proc. Artif. Intell.*, vol. 232, 2016, pp. 20–42.

[13] L. C. T. Bezerra, M. López-Ibáñez, and T. Stützle, "Automatic design of evolutionary algorithms for multi-objective combinatorial optimization," in *Proc. 13th Int. Conf. Parallel Problem Solving Nat. (PPSN)*, Sep. 2014, pp. 508–517.

[14] L. Xu, H. H. Hoos, and K. Leyton-Brown, "Hydra: Automatically configuring algorithms for portfolio-based selection," in *Proc. 24th AAAI Conf. Artif. Intell. (AAAI)*, Jul. 2010, pp. 210–216.

[15] 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.

[16] J. Seipp, S. Sievers, M. Helmert, and F. Hutter, "Automatic configuration of sequential planning portfolios," in *Proc. 29th AAAI Conf. Artif. Intell. (AAAI)*, Jan. 2015, pp. 3364–3370.

[17] 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.

[18] S. Liu, K. Tang, and X. Yao, "Automatic construction of parallel portfolios via explicit instance grouping," in *Proc. 33rd AAAI Conf. Artif. Intell. (AAAI)*, Jan. 2019, pp. 1560–1567.

[19] H. H. Hoos, "Programming by optimization," *Commun. ACM*, vol. 55, no. 2, pp. 70–80, 2012.

[20] J. N. Hooker, "Testing heuristics: We have it all wrong," *J. Heuristics*, vol. 1, no. 1, pp. 33–42, 1995.

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

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

[23] 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.

[24] S. Wøhlk and G. Laporte, "A fast heuristic for large-scale capacitated arc routing problems," *J. Oper. Res. Soc.*, vol. 69, no. 12, pp. 1877–1887, 2018.

[25] 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.

[26] C. P. Gomes and B. Selman, "Algorithm portfolios," *Artif. Intell.*, vol. 126, nos. 1–2, pp. 43–62, 2001.

[27] K. Asanovic *et al.*, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[28] J. R. Rice, "The algorithm selection problem," *Adv. Comput.*, vol. 15, pp. 65–118, May 2008.

[29] 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.

[30] 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.

[31] H. Han, W. Lu, L. Zhang, and J. Qiao, "Adaptive gradient multiobjective particle swarm optimization," *IEEE Trans. Cybern.*, vol. 48, no. 11, pp. 3067–3079, Nov. 2018.

[32] M. Asafuddoula, H. K. Singh, and T. Ray, "An enhanced decomposition-based evolutionary algorithm with adaptive reference vectors," *IEEE Trans. Cybern.*, vol. 48, no. 8, pp. 2321–2334, Aug. 2018.

[33] Y. Hua, Y. Jin, and K. Hao, "A clustering-based adaptive evolutionary algorithm for multiobjective optimization with irregular Pareto fronts," *IEEE Trans. Cybern.*, vol. 49, no. 7, pp. 2758–2770, Jul. 2019.

[34] R. Battiti, M. Brunato, and F. Mascia, Eds., *Reactive Search and Intelligent Optimization*. New York, NY, USA: Springer, 2008.

[35] L. Ke, Q. Zhang, and R. Battiti, "MOEA/D-ACO: A multiobjective evolutionary algorithm using decomposition and antcolony," *IEEE Trans. Cybern.*, vol. 43, no. 6, pp. 1845–1859, Dec. 2013.

[36] E. K. Burke *et al.*, "Hyper-heuristics: A survey of the state of the art," *J. Oper. Res. Soc.*, vol. 64, no. 12, pp. 1695–1724, 2013.

[37] Z. Ren, H. Jiang, J. Xuan, Y. Hu, and Z. Luo, "New insights into diversification of hyper-heuristics," *IEEE Trans. Cybern.*, vol. 44, no. 10, pp. 1747–1761, Oct. 2014.

[38] S. Nguyen, M. Zhang, and K. C. Tan, "Surrogate-assisted genetic programming with simplified models for automated design of dispatching rules," *IEEE Trans. Cybern.*, vol. 47, no. 9, pp. 2951–2965, Sep. 2017.

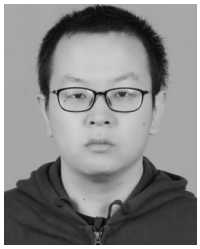
[39] K. Smith-Miles, "Cross-disciplinary perspectives on meta-learning for algorithm selection," *ACM Comput. Survey*, vol. 41, no. 1, pp. 1–25, 2008.

[40] J. I. van Hemert, "Evolving combinatorial problem instances that are difficult to solve," *Evol. Comput.*, vol. 14, no. 4, pp. 433–462, 2006.

[41] K. Smith-Miles, J. I. van Hemert, and X. Y. Lim, "Understanding TSP difficulty by learning from evolved instances," in *Proc. 4th Int. Conf. Learn. Intell. Optim. (LION)*, Jan. 2010, pp. 266–280.

[42] K. Smith-Miles and J. I. van Hemert, "Discovering the suitability of optimisation algorithms by learning from evolved instances," *Ann. Math. Artif. Intell.*, vol. 61, no. 2, pp. 87–104, 2011.

[43] O. Mersmann, B. Bischl, H. Trautmann, M. Wagner, J. Bossek, and F. Neumann, "A novel feature-based approach to characterize algorithm performance for the traveling salesperson problem," *Ann. Math. Artif. Intell.*, vol. 69, no. 2, pp. 151–182, 2013.
[44] S. Nallaperuma, M. Wagner, and F. Neumann, "Parameter prediction based on features of evolved instances for ant colony optimization and the traveling salesperson problem," in *Proc. 13th Int. Conf. Parallel Problem Solving Nat. (PPSN)*, Sep. 2014, pp. 100–109.
[45] W. Gao, S. Nallaperuma, and F. Neumann, "Feature-based diversity optimization for problem instance classification," in *Proc. 14th Int. Conf. Parallel Problem Solving Nat. (PPSN)*, Sep. 2016, pp. 869–879.
[46] Y. Malitsky, M. Merschformann, B. O'Sullivan, and K. Tierney, "Structure-preserving instance generation," in *Proc. 10th Int. Conf. Learn. Intell. Optim. (LION)*, May 2016, pp. 123–140.
[47] J. Branke and C. W. Pickardt, "Evolutionary search for difficult problem instances to support the design of job shop dispatching rules," *Eur. J. Oper. Res.*, vol. 212, no. 1, pp. 22–32, 2011.
[48] J. H. Moreno-Scott, J. C. Ortiz-Bayliss, H. Terashima-Marín, and S. E. Conant-Pablos, "Challenging heuristics: Evolving binary constraint satisfaction problems," in *Proc. 12th Genet. Evol. Comput. Conf. (GECCO)*, Jul. 2012, pp. 409–416.
[49] I. Amaya, J. C. Ortiz-Bayliss, S. E. Conant-Pablos, H. Terashima-Marín, and C. A. Coello Coello, "Tailoring instances of the 1D bin packing problem for assessing strengths and weaknesses of its solvers," in *Proc. 15th Int. Conf. Parallel Problem Solving Nat. (PPSN)*, Sep. 2018, pp. 373–384.
[50] I. J. Goodfellow *et al.*, "Generative adversarial nets," in *Proc. 27th Annu. Conf. Adv. Neural Inf. Process. Syst. (NIPS)*, Dec. 2014, pp. 2672–2680.
[51] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel, "InfoGAN: Interpretable representation learning by information maximizing generative adversarial nets," in *Proc. 29th Annu. Conf. Adv. Neural Inf. Process. Syst. (NIPS)*, Dec. 2016, pp. 2172–2180.
[52] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, "Spectral normalization for generative adversarial networks," in *Proc. 6th Int. Conf. Learn. Represent. (ICLR)*, Apr. 2018, pp. 1–26.
[53] H. Zhu *et al.*, "Single-image dehazing via compositional adversarial network," *IEEE Trans. Cybern.*, early access, Dec. 31, 2019, doi: 10.1109/TCYB.2019.2955092.
[54] J. Lu, A. Kannan, J. Yang, D. Parikh, and D. Batra, "Best of both worlds: Transferring knowledge from discriminative learning to a generative visual dialog model," in *Proc. 30th Annu. Conf. Adv. Neural Inf. Process. Syst. (NIPS)*, Dec. 2017, pp. 314–324.
[55] X. Chen, C. Xu, X. Yang, L. Song, and D. Tao, "Gated-GAN: Adversarial gated networks for multi-collection style transfer," *IEEE Trans. Image Process.*, vol. 28, no. 2, pp. 546–560, Feb. 2019.
[56] Z. Zhong, J. Li, D. A. Clausi, and A. Wong, "Generative adversarial networks and conditional random fields for hyperspectral image classification," *IEEE Trans. Cybern.*, early access.
[57] X. Peng, J. Feng, S. Xiao, W.-Y. Yau, J. T. Zhou, and S. Yang, "Structured autoencoders for subspace clustering," *IEEE Trans. Image Process.*, vol. 27, no. 10, pp. 5076–5086, Oct. 2018.
[58] X. Peng, Z. Huang, J. Lv, H. Zhu, and J. T. Zhou, "COMIC: Multi-view clustering without parameter selection," in *Proc. 36th Int. Conf. Mach. Learn. (ICML)*, Jun. 2019, pp. 5092–5101.
[59] Z. Dai, A. Almahairi, P. Bachman, E. H. Hovy, and A. C. Courville, "Calibrating energy-based generative adversarial networks," in *Proc. 5th Int. Conf. Learn. Represent. (ICLR)*, Apr. 2017, pp. 1–17.
[60] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, Aug. 2017, pp. 214–223.
[61] C. Wang, C. Xu, X. Yao, and D. Tao, "Evolutionary generative adversarial networks," *IEEE Trans. Evol. Comput.*, vol. 23, no. 6, pp. 921–934, Dec. 2019.
[62] 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.
[63] T. Back, D. B. Fogel, and Z. Michalewicz, Eds., *Handbook of Evolutionary Computation*. Bristol, U.K.: IOP, 1997.
[64] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions—I," *Math. Program.*, vol. 14, no. 1, pp. 265–294, 1978.
[65] K. Helsgaun, "An effective implementation of the Lin–Kernighan traveling salesman heuristic," *Eur. J. Oper. Res.*, vol. 126, no. 1, pp. 106–130, 2000.
[66] A. Biere, "Lingeling and friends entering the SAT challenge 2012," in *Proc. Challenge Solver Benchmark Descriptions*, 2012, pp. 33–34.
[67] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. (2006). *Concorde TSP Solver*. [Online]. Available: http://www.math.uwaterloo.ca/tsp/concorde.html
[68] 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.
[69] A. Wotzlaw, A. van der Grinten, E. Speckenmeyer, and S. Porschen, "pFolioUZK: Solver description," in *Proc. SAT Challenge Solver Benchmark Descriptions*, 2012, p. 45.
[70] A. Biere, "Splatz, lingeling, plingeling, treengeling, YalSAT entering the SAT competition 2016," in *Proc. Competition Solver Benchmark Descriptions*, 2016, pp. 44–45.
[71] M. Crepinsek, S. Liu, and M. Mernik, "Exploration and exploitation in evolutionary algorithms: A survey," *ACM Comput. Survey*, vol. 45, no. 3, pp. 1–33, 2013.
[72] K. Tang, P. Yang, and X. Yao, "Negatively correlated search," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 3, pp. 542–550, Mar. 2016.



Shengcai Liu (Student Member, IEEE) received the B.Eng. degree in computer science and technology from the University of Science and Technology of China, Hefei, China, in 2014, where he is currently pursuing the Ph.D. degree with the School of Computer Science and Technology.

His research interests include combinatorial optimization, automatic parameter tuning, automatic solver design, and automated machine learning.

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 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 70 journal papers and more than 80 conference papers. According to Google Scholar, his publications have received more than 8000 citations and the H-index is 42. His major research interests include evolutionary computation, machine learning, and their applications.

Prof. Tang received 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.

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, Beijing, 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 current research interests include evolutionary computation, machine learning, and their real-world applications, especially to software engineering.

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