Automated Test Suite Generation for Software Product Lines Based on Quality-Diversity Optimization

YI XIANG, HAN HUANG, and SIZHE LI, South China University of Technology, China
MIQING LI, University of Birmingham, UK
CHUAN LUO, Beihang University, China
XIAOWEI YANG, South China University of Technology, China

A Software Product Line (SPL) is a set of software products that are built from a variability model. Real-world SPLs typically involve a vast number of valid products, making it impossible to individually test each of them. This arises the need for automated test suite generation, which was previously modeled as either a single-objective or a multi-objective optimization problem considering only objective functions. This article provides a completely different mathematical model by exploiting the benefits of Quality-Diversity (QD) optimization that is composed of not only an objective function (e.g., *t*-wise coverage or test suite diversity) but also a user-defined behavior space (e.g., the space with test suite size as its dimension). We argue that the new model is more suitable and generic than the two alternatives because it provides at a time a large set of diverse (measured in the behavior space) and high-performing solutions that can ease the decision-making process. We apply MAP-Elites, one of the most popular QD algorithms, to solve the model. The results of the evaluation, on both realistic and artificial SPLs, are promising, with MAP-Elites significantly and substantially outperforming both single- and multi-objective approaches, and also several state-of-the-art SPL testing tools. In summary, this article provides a new and promising perspective on the test suite generation for SPLs.

CCS Concepts: • **Software and its engineering → Search-based software engineering**; **Empirical software validation;**

Additional Key Words and Phrases: Software Product Line, automated test suite generation, Quality-Diversity (QD) optimization

ACM Reference format:
Yi Xiang, Han Huang, Sizhe Li, Miqing Li, Chuan Luo, and Xiaowei Yang. 2023. Automated Test Suite Generation for Software Product Lines Based on Quality-Diversity Optimization. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 46 (December 2023), 52 pages.
https://doi.org/10.1145/3628158

This work was supported by the National Natural Science Foundation of China (61906069, 62276103, 62202025), the Science and Technology Program of Guangzhou (202002030355, 201802010007), the Guangdong Basic and Applied Basic Research Foundation (2214050004299, 2019A1515011411, 2019A1515011700), the Guangdong Province Key Area R&D Program (2020B0303300001, 2018B010109003), Fundamental Research Funds for the Central Universities (2020ZYGXZR014), and the Natural Science Research Project of Education Department of Guizhou Province (QJJ2023061, QJJ2023012).
Authors' addresses: Y. Xiang, H. Huang (Corresponding author), S. Li, and X. Yang (Corresponding author), School of Software Engineering, South China University of Technology, Guangzhou, China; e-mails: gzhuxiang_yi@163.com, {hhan, xwyang}@scut.edu.cn, li_sizhe2000@163.com; M. Li, School of Computer Science, University of Birmingham, Edgbaston, Birmingham, UK; e-mail: limitsing@gmail.com; C. Luo, School of Software, Beihang University, Beijing, China; e-mail: chuanluo@buaa.edu.cn.

1049-331X/2023/12-ART46 $15.00
https://doi.org/10.1145/3628158

1 INTRODUCTION

A **Software Product Line (SPL)** [16] is a family of related software products, each of which provides a different combination of features, with a *feature* being commonly defined as an increment in system functionality [61]. **Feature Models (FMs)** [8, 9] are often used to define the allowed combinations of features and so valid products. Real-world SPLs typically involve a vast number of valid products, which makes it infeasible, or at least inefficient, to individually test each one of them [45]. It has led to the need for adequate and scalable SPL testing techniques.

In recent years, there has been a growing interest in SPL testing, which is reflected by several systematic mapping studies and surveys on this topic [22, 31, 52, 53, 61, 62]. One central problem in SPL testing is to automatically generate a test suite or a sample (i.e., a set of valid products) that is expected to reveal as many faults as possible. This problem is referred to as an *automated test suite generation* problem, which is also frequently known as a *sample generation* problem. Since testers normally have no access to information on the actual faults in the products before the testing begins, the generation of test suites is often guided by some *objective functions* that capture properties of a good test suite regarding exposing faults. The objective function maps a test suite to a value that quantifies how "good" a test suite is according to these properties. For example, a fault might be triggered by interactions of a pair of features. Thus, we might want to generate a test suite that maximizes the number of such interactions (i.e., pairwise/2-wise coverage), then potentially reveals as many faults as possible. A number of objective/fitness functions have been proposed in the literature (see, e.g., pairwise coverage [62], test suite size [43, 59, 60], test suite cost [43, 94], and dissimilarity [2, 42]).

In previous work, the automated test suite generation has been modeled as either a single-objective optimization problem [1, 29, 37, 41, 42, 50, 66, 67], a multi-objective optimization problem [24, 43, 60, 69, 77, 89, 93], or a many-objective[1] optimization problem [33, 34, 45, 94, 95]. Due to the large, complex, and discrete search spaces, search-based optimization techniques, especially those developed based on evolutionary computation, have been widely and successfully applied to the automated test suite generation problem (see, e.g., [29, 33, 34, 41, 42, 45, 60, 77, 89, 94, 95]). Regarding this topic, a recent mapping study was provided by Lopez-Herrejon et al. [61].

However, single-objective test suite generation approaches provide each time only one solution (i.e., a test suite in our context) to software engineers, which may mismatch the economical and technological constraints of their testing scenarios [61]. For example, according to Johansen et al. [47], a test suite with 480 products is required to guarantee full pairwise coverage for the Linux kernel FM (2.6.28.6-icse11) [88]. Unfortunately, testing 480 products might be unaffordable due to limited test budgets in practice. Furthermore, software engineers might have different test preferences, and they may even change them over time. For example, one may emphasize test quality (e.g., achieving full coverage of feature combinations) over test efficiency, and another one may want to quickly test just a certain number of products at the sacrifice of test quality. To meet various preferences, one needs to independently run the single-objective approach multiple times with different settings (e.g., different test suite sizes). This is definitely computationally inefficient.

Lopez-Herrejon et al. [61] argued that searching for a set of solutions in a single run can overcome the preceding disadvantages of single-objective approaches. Indeed, multiple solutions can help software engineers make informed decisions (i.e., selecting the test suite that best matches their economical and technological constraints). Given the fact that there exist scenarios where software engineers do face tradeoffs among multiple and often conflicting objectives, they suggested formalizing SPLs testing as a multi-objective optimization problem. Focusing

[1]In the evolutionary multi-objective community, many-objective optimization is a special case of multi-objective optimization, where four or more objectives are simultaneously optimized [56].

also on generating a set of solutions, this article provides an entirely new perspective—the test suite generation is modeled as a **Quality-Diversity (QD)** optimization problem [80]. The QD optimization is a new paradigm for stochastic optimization [13], whose aim is to search for a large set of *diverse* but *high-performing* solutions rather than only the best one, like in single-objective optimization, or the Pareto front, like in multi-objective optimization. The prominent characteristic of QD optimization is that diversity is measured in the customized *behavior space*, which allows users to flexibly specify the types of solutions that they are most interested in. The goal of QD optimization is to find the highest-performing solution for each point in the behavior space (possibly discretized). Then users can pick the high-performing solutions that they deem as the most interesting according to their own knowledge.

The QD optimization seems to well fit the nature of automated test suite generation for SPLs due to at least the following reasons. First, a set of diverse test suites is required in SPL testing [61], whereas QD optimization inherently produces a collection of diverse and high-performing solutions. That is to say, QD optimization well matches the goal of SPL testing. Second, test suite generation for SPLs needs to take into account some dimensions that are related to technical and economical constraints [61], as well as software engineers' preferences. This can be easily captured and implemented by defining the behavior space in QD optimization. For example, software engineers may be interested in how the pairwise coverage varies with respect to the size of a test suite. Then, they can define the size as one dimension of the behavior space. Put simply, QD optimization makes easy and straightforward the decision-making process (i.e., selecting the best test suite to apply). Last but not the least, there have been some high-performing off-the-shelf QD optimization algorithms that can be adapted in the context of SPL testing, such as **Novelty Search (NS)**[2] [54], **Novelty Search with Local Competition (NSLC)** [55], and **multi-dimensional archive of phenotypic elites (MAP-Elites)** [72]. These algorithms have achieved a great success in evolutionary robotics [18, 21, 55, 72, 81] and video games [12, 28, 38, 58]. Previous work also found them to be effective in **Search-Based Software Engineering (SBSE)** [40], particularly search-based software testing (see, e.g., [11, 30, 68, 82, 101, 102]).

Despite of the preceding facts, there has been almost no work that fully exploits the benefits of QD optimization in automated test suite generation in the context of SPL testing, and this article takes the first step in this regard. We model, for the first time, the test suite generation for SPLs as a QD optimization problem and use MAP-Elites [72] to solve it. To the best of our knowledge, MAP-Elites has never been applied to SPL testing before. The most relevant work [96] is probably one of our previous works on SPL testing. In that work, NS was adopted to generate a single test suite with diverse test cases in the decision/parameter space. In contrast, this article works in the test suite level (i.e., generating a set of diverse test suites). We evaluated our approach on 105 FMs that were collected from four related papers [7, 42, 45, 96]. These FMs have been widely used as benchmarks to assess SPL testing techniques [7, 42, 45, 49, 60, 66, 77, 78, 96]. Our results indicate that MAP-Elites shows significant and substantial improvements over single-objective and multi-objective approaches, as well as state-of-the-art SPL testing tools in terms of generating a set of diverse and high-performing test suites. Compared with the recent NS-based test suite generation algorithm [96], MAP-Elites is able to disclose more faults on the majority of the FMs under study.

This article makes the following main contributions:

- We model, for the first time, the test suite generation for SPLs as a QD optimization problem. This is a new perspective on modeling the test suite generation problem, and the goal of this model is to generate a set of diverse and high-performing test suites. To instantiate the model, the fitness function is defined as the t-wise coverage or test suite diversity, and

[2]NS completely ignores the objective but instead searches only for behavioral diversity.

the behavior space is a one-dimensional space with test suite size being its dimension. It is worth mentioning that the objective function and the behavior space can be customized by software engineers based on their own testing contexts. Compared with the single-objective model, the QD model can better support the decision-making process, as it provides multiple solutions at a time rather than only a single one (an illustration is given in Section 2.3). Compared with the multi-objective model, the QD model is more generic, as it does not impose conflicts between the objective function and the measure functions (that define the behavior space) [100]. In multi-objective optimization, however, objective functions, or at least some of them, are typically required to be conflicting.[3] Therefore, software engineers need to be careful when defining the objectives.

- We use MAP-Elites, one of the most famous QD algorithms, to solve the QD-based test suite generation model. It is found that MAP-Elites performs significantly better than the vanilla approach where a single-objective genetic algorithm is independently run multiple times to generate the optimal or near-optimal solution for each point in the behavior space. We show that the success of MAP-Elites is attributed to its inherent mechanism that enables information sharing [13]. It is also found that MAP-Elites significantly outperforms NSGA-II[4] [23], one of the most widely used **Multi-Objective Evolutionary Algorithms (MOEAs)**. We argue that the superiority of MAP-Elites over NSGA-II is achieved because MAP-Elites performs local competition instead of global competition (as in NSGA-II). Global competition has the risk of missing some points in the behavior space [55], and more detailed discussions are available in Section 6.2.
- We compare MAP-Elites with the state-of-the-art t-wise testing tools (i.e., YASA [50], IncLing [1], and SamplingCA [67]), and the recently proposed search-based testing algorithms (i.e., GrES (grid-based evolution strategy) [45] and the NS-based algorithm [96]). Our results show that MAP-Elites performs better than YASA and IncLing but worse than SamplingCA in terms of generating multiple high-performing test suites (a possible explanation is given in Section 6.3). Moreover, MAP-Elites outperforms GrES in reducing the size of test suites generated by the three t-wise tools. Finally, it significantly outperforms the NS-based algorithm with respect to both test suite diversity and fault detection rate. This is not surprising, as we find by performing correlation analysis that test suite diversity has, in most cases, a significantly positive correlation with the fault detection rate. For the sake of reproducibility, we make source codes, along with the used FMs, publicly available at https://github.com/gzhuxiangyi/SPLTestingMAP. In addition, supplementary materials and raw data are uploaded to https://doi.org/10.5281/zenodo.7805017.

The remainder of the article is structured as follows. In Section 2, we review background materials. Section 3 describes the QD optimization model for SPL test suite generation, whereas Section 4 outlines the MAP-Elites that is used to solve the model. In Section 5, we give the experimental design, followed by the results of the experiments and discussions presented in Section 6. Subsequently, Section 7 discusses practical implications of the proposed approach, and Section 8 reviews related work on SPL testing as well as applications of QD optimization. Finally, we draw conclusions and discuss possible directions for future studies in Section 9.

[3]When objectives are not conflicting, multi-objective optimization algorithms can also be applied but may not be efficient enough.

[4]According to Lopez-Herrejon et al. [61], the t-wise coverage (more precisely pairwise coverage) and test suite size are conflicting. Thus, our QD model can be transformed into a bi-objective problem. However, this transformation is not always possible because objectives in multi-objective optimization should be conflicting in general. After transformation, MOEAs (e.g., NSGA-II) can be adopted to solve the problem.

2 BACKGROUND

In this section, we provide the basic background on the two topics that crosscut the article: SPL testing and QD optimization.

2.1 SPL Testing

In our context, the SPL testing algorithm uses an FM [8] as input, and outputs a set of test suites, each of which is composed of multiple test cases (i.e., products). We present the following key concepts and notations on which SPL testing terminology is defined.

Definition 2.1. A *feature model* is a tuple $(\mathcal{F}, C)$ [42], where $\mathcal{F} = \{f_1, \ldots, f_n\}$ represents a set of n features (Boolean variables), and $C = \{c_1, \ldots, c_m\}$ is a set of m constraints among these features. C is satisfied if and only if all c_i $(i = 1, \ldots, m)$ are satisfied.

According to Batory [8], an FM can be easily translated into a Boolean formula in **Conjunctive Normal Form (CNF)**. Therefore, in **Satisfiability (SAT)** terminology, each c_i $(i = 1, \ldots, m)$ is actually a *clause*, which is a disjunction of several *literals*.

Definition 2.2. A *product* is a set $p = \{\pm f_1, \ldots, \pm f_n\}$, where $+f_i$ and $-f_i$ indicate that the feature f_i is selected and deselected, respectively. Note that a feature can be either selected or deselected. Moreover, a product is said to be valid if and only if all the FM constraints are satisfied. Otherwise, it is called an *invalid product*.

In the context of SPL testing, a test case is a valid product, and a test suite is composed of multiple test cases. Formally, we have the following definition.

Definition 2.3. A *test suite* of an SPL is a set $TS = \{tc_1, \ldots, tc_N\}$, where each tc_i $(i = 1, \ldots, N)$ is a *test case* (i.e., valid product), and N denotes the test suite size.

The t-wise coverage is the most popular and important criterion to evaluate the quality of a test suite. We list the following definitions related to this criterion.

Definition 2.4. Suppose $t \leq n$ and suppose also that $f_{\pi_1}, \ldots, f_{\pi_t}$ are t distinct features selected from $\mathcal{F}$. The set $\{\pm f_{\pi_1}, \ldots, \pm f_{\pi_t}\}$ is called a *t-set*, which essentially represents a partially configured product [37, 62].

A t-set is called a *valid t-set* if it satisfies the constraint set C of the FM. A valid t-set is covered by a test case tc if it is a subset of tc. It is worth mentioning that invalid t-sets do not need to be covered [2]. For SPL testing, the t-wise coverage is probably the most widely used test criterion. Mathematically, we have the following definition.

Definition 2.5. The *t-wise coverage* of a test suite $TS = \{tc_1, \ldots, tc_N\}$ is defined as the following ratio:

$$\frac{|\cup_{i=1}^{N} \mathcal{VT}_{tc_i}|}{|\mathcal{VT}_{fm}|} \times 100\%, \tag{1}$$

where $\mathcal{VT}_{fm}$ is the set of all the valid t-sets for a given FM; $\mathcal{VT}_{tc_i}$ denotes the set of t-sets covered by the test case tc_i, and $|\cdot|$ returns the cardinality of a set.

Finally, we give the definition of covering arrays.

Definition 2.6. A *t-wise covering array* is a particular test suite by which all valid t-sets are covered [17, 48]. Typically, small-sized covering arrays are desired, as they could reduce the testing cost.

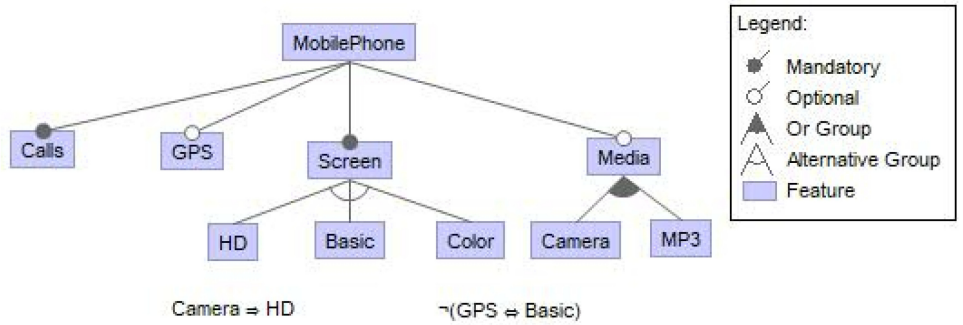


Fig. 1. A mobile phone FM.

Table 1. All Test Cases for the Mobile Phone FM

ID	MobilePhone	Calls	GPS	Screen	Basic	Color	HD	Media	Camera	MP3
tc_1	✓	✓	✓	✓		✓				
tc_2	✓	✓	✓	✓		✓		✓		✓
tc_3	✓	✓	✓	✓			✓	✓		✓
tc_4	✓	✓	✓	✓			✓			
tc_5	✓	✓	✓	✓			✓	✓	✓	
tc_6	✓	✓	✓	✓			✓	✓	✓	✓
tc_7	✓	✓		✓	✓					
tc_8	✓	✓		✓	✓			✓		✓

Selected features are ticked (✓), and unselected features are empty.

Figure 1 shows the FM of a simplified mobile phone product line. We use it as an illustrative example to help readers understand the preceding definitions. This FM has 10 features, $\mathcal{F}$={MobilePhone, Calls, GPS, Screen, Basic, Color, HD, Media, Camera, MP3}, and 22 CNF constraints after being transformed into a Boolean formula. The total number of valid products (or test cases) represented by this model is 8, and all of them are listed in Table 1. This model has 116 valid 2-sets. For example, {MobilePhone, Calls} is a valid one, whereas {GPS, Basic} is invalid because it violates the constraint "¬ (GPS ⇔ Basic)." The 2-wise coverage of the test suite $\{tc_6, tc_7\}$ is 72.41%, and that of $\{tc_1, tc_2, tc_4, tc_5, tc_6, tc_7, tc_8\}$ is 100%. Hence, the latter is a 2-wise covering array.

2.2 QD Optimization

As mentioned in Section 1, QD optimization [13, 80] is a new paradigm of stochastic optimization. The QD problem is defined over a search space with an objective/fitness function to optimize, and a few user-defined measure functions to span [100]. The measure function outputs are often referred to as behavior characterization [54], behavior descriptors [73], or features [72], which are a numerical vector that characterizes the overall behavior of a solution. Accordingly, the space formed is known as a behavior/feature space. For example, for robot morphologies [72], the fitness function could be how fast the robot is; the behaviors of interest could be its height, weight, and energy consumption, leading to a three-dimensional behavior space. The goal of QD optimization is to find the highest-performing solution for each point in the behavior space. Because this type of algorithm illuminates the fitness potential of each area of the behavior space, including tradeoffs between quality and behaviors of interest, it is also called an *illumination algorithm* [72].

ALGORITHM 1: General framework of MAP-Elites

Input: M (number of initial solutions)
Output: $\mathcal{A}$ (grid/archive)
1 Randomly initialize M solutions and put them into the grid $\mathcal{A}$;
2 **while** *termination condition is not met* **do**
3 Uniformly select solutions from $\mathcal{A}$;
4 Create new solutions based on the selected one(s);
5 Evaluate new solutions and potentially add them into $\mathcal{A}$;
6 **end**
7 **return** $\mathcal{A}$

NSLC [55] and MAP-Elites [72] are two of the most famous QD algorithms, and both of them are based on evolutionary algorithms. In particular, MAP-Elites is most widely used due to that it is conceptually simple and easy to implement. MAP-Elites discretizes the behavior space into a grid (also called an *archive* or a *map*), and its goal is to fill each cell of this grid with the highest-performing solutions. The following is a brief introduction to the general framework of this algorithm (Algorithm 1). It starts with a random initialization of a fixed number of solutions (M), which are evaluated and then placed into a grid. After initialization, MAP-Elites enters the main loop where each iteration is composed of uniform selection of a solution from the grid, creation of new solutions (via crossover and mutation), evaluation of fitness and behavior descriptors for the new solutions, and potential addition of them into the grid. A solution is added into the grid if it either occupies an empty cell or improves the current solution in the same cell. As the search proceeds, more cells get filled and better solutions are inserted into the grid. The preceding grid maintenance mechanism suggests that MAP-Elites performs local competition instead of global competition as in traditional evolutionary algorithms. That is to say, a solution competes for fitness (or performance) only with other solutions within the same cell.

2.3 An Example Scenario

Let us now motivate the usefulness of QD optimization for SPL testing with a simple and illustrative example. Consider the scenario where 2-wise coverage is the fitness function to maximize, and test suite size is the measure function to span. Figure 2 shows the information on test suites returned by MAP-Elites on the mobile phone FM. We now explain how a set of diverse and high-performing solutions can aid the decision-making process. As already well discussed in the work of Lopez-Herrejon et al. [61], selecting from multiple test suites allows decision makers (i.e., software engineers) to make informed decisions to best match the economical and technological constraints of their own testing context. Consider the following:

- Suppose that the first decision maker emphasizes test quality over test efficiency. Clearly, from Figure 2, the best test suite he/she will choose is TS_6, as it is the minimal test suite that guarantees full 2-wise coverage.
- Suppose that the second decision maker just wants to get a pairwise coverage beyond 95%. Again, from the information provided in Figure 2, he/she will choose TS_4, as it is the minimal test suite that satisfies the preceding technical constraint.
- Suppose that the third decision maker can only test four products due to economical constraints. Once more, using the information in Figure 2, he/she will choose TS_3 and will know that the 2-wise coverage can reach 93.97%.

According to the preceding discussions, providing a set of diverse and high-performing solutions can help decision makers choose the right test suite to use in practice when facing

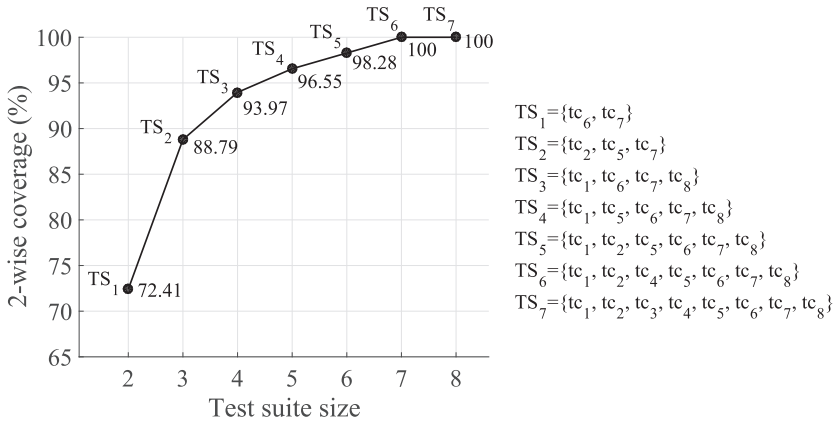


Fig. 2. Information on test suites returned by MAP-Elites on the mobile phone FM. For test suites from TS_3 to TS_7, as can be seen, previous test cases selected for smaller test suites are always contained in the larger test suites. However, it is not true for the test suites from TS_1 to TS_3.

with various technical and economical constraints. Moreover, when the testing context changes, decision makers can easily adapt to changes by turning to other alternatives. This is in sharp contrast with single-objective optimization that can only provide a single test suite. When this test suite mismatches the testing context, no alternatives are available, and thus software engineers need to rerun the single-objective algorithm to get another one. However, this is likely ineffective due to the absence of information sharing; more detailed discussions will be given in Section 6.1.3. Although multi-objective optimization with the two objectives—maximizing 2-wise coverage and minimizing test suite size—can also generate a set of test suites as QD optimization in our example scenario, it is less generic because it requires that the objective functions are conflicting. Furthermore, due to the global competition, multi-objective optimization has the risk of missing some areas of the behavior space (i.e., missing test suites with certain sizes in our case), and this issue will be investigated in depth in Section 6.2. In summary, QD optimization can be used as a generic, flexible, and effective alternative to single- and multi-objective optimization in generating multiple and high-performing test suites in the context of SPL testing.

3 TEST SUITE GENERATION AS A QD OPTIMIZATION PROBLEM

Without loss of generality, we assume that the fitness function is to be maximized. Let $\mathcal{B}$ be the behavior space, and the goal in QD optimization is to find, for each point $\boldsymbol{b} \in \mathcal{B}$, the solution $\boldsymbol{x}^*$ with the maximum fitness value. Mathematically,

$$\begin{aligned} &\forall \boldsymbol{b} \in \mathcal{B},\ \boldsymbol{x}^* = \arg\max_{\boldsymbol{x}} f(\boldsymbol{x}) \\ &\quad s.t.\ \boldsymbol{b}(\boldsymbol{x}) = \boldsymbol{b}, \end{aligned} \tag{2}$$

where $\boldsymbol{b}(\boldsymbol{x})$ returns the behavior descriptor of $\boldsymbol{x}$. The constraint in the preceding model is to restrict the search in a particular region where all solutions have the same behavior descriptor $\boldsymbol{b}$. According to Chatzilygeroudis et al. [14], the QD optimization problem can be viewed as a set of optimizations constrained by the points in $\mathcal{B}$, which we call *QD subproblems* hereafter. In the following sections, we will show how the preceding QD optimization model is applied to the SPL test suite generation problem by instantiating its main components, including solution representation and encoding (Section 3.1), definition of behavior spaces (Section 3.2), and definition of fitness functions (Section 3.3).

3.1 Test Suites as Solutions

In Model (2), a solution $\boldsymbol{x}$ represents a test suite $TS = \{tc_1, \ldots, tc_N\}$, where N denotes the size of TS, or the number of test cases. Each test case (i.e., a valid product of an SPL) is represented by n bits (with n being the number of features). Specifically, a value of 1 means the corresponding feature is selected in the test case, whereas a value of 0 means the feature being deselected. Note that test cases are generated by using two tools: SamplingCA [67] and PLEDGE [44] (which internally uses a modified version of SAT4J [42]). Hence, a test suite with N test cases is encoded as a binary string of length $N \times n$. It is worth mentioning that this binary string encoding has been widely adopted in prior work [45, 77] in the context of SPL testing.

3.2 Definition of Behavior Space $\mathcal{B}$

The behavior space is formed by one or more behavior descriptors of interest. A behavior descriptor typically describes how the solution solves a problem. Behaviors can be either defined based on domain knowledge or learned automatically before or during the search process [19, 39, 74, 75]. In this article, we define $\mathcal{B}$ as a one-dimensional space by choosing test suite size as the only behavior. In fact, the test suite size is an important feature characterizing a test suite. Larger test suites generally lead to higher t-wise coverage or fault detection rate, but come with higher test costs, and vice versa. Hence, the size of a test suite plays a vital role in balancing between test quality and test cost. Moreover, because test budgets and test scenarios vary in practice, testers may prefer test suites with a particular size. In fact, test suite size has been widely used as a fitness function in SPL testing [33, 34, 43, 45, 60, 89]. Notice that the choice of behavior descriptors is relatively flexible. As outlined in the future work, other behavior descriptors can also be considered.

Notice that values of the test suite size can be any integer larger than 0. If the behavior space is unbounded, computational burdens would become prohibitively high. Following the common practice in QD optimization studies, we impose lower and upper bounds on the behavior space $\mathcal{B}$. More specifically, values of the test suite size is restricted to the interval $[lb, ub]$. Thus, $\mathcal{B}$ is discretized into $(ub - lb)/\delta$ cells, where δ denotes the discretization granularity with 1 being the default value. Then, we will search for the highest-performing solution for each of the cells in the behavior space.

Finally, one thing to note is that lb, ub, and δ are manually specified based on either software engineers' preference or available resources. In our experimental study, we set ub and lb to different values in different experiments. More details and explanations will be given later in Section 5.3.

3.3 Fitness Functions

The fitness value $f(\boldsymbol{x})$ quantifies how well the solution $\boldsymbol{x}$ solves a problem. For SPL testing, t-wise coverage, defined by Equation (1), is probably the most widely used fitness function. For a test suite $TS = \{tc_1, \ldots, tc_N\}$, we use $f_{cov}(TS)$ to denote its t-wise coverage. Note that when t gets higher or FM gets larger, $f_{cov}(TS)$ becomes more computationally expensive. The reason is that to compute $f_{cov}(TS)$, all valid t-sets should be worked out. However, this is not easy because the total number of all possible valid t-sets (i.e., $|\mathcal{VT}_{fm}|$) increases rapidly with respect to t and the size of the FM. In fact, $|\mathcal{VT}_{fm}|$ can be as large as $2^t \cdot C_n^t$ [51], where n is the number of features. Therefore, $f_{cov}(TS)$ is primarily adopted in the cases where FMs are small and t is low (e.g., $t = 2$ or $t = 3$). In particular, 2-wise coverage (also known as pairwise coverage) has been extensively used in related work in SPL testing [64]. Notably, Hierons et al. [45] considered nine fitness functions, among which 2-wise coverage was deemed more important than others. This was motivated by the fact that 2-wise coverage can be seen as the main aim in test suite generation. In this article, we also focus on 2-wise coverage.

For large FMs (e.g., with more than 100 features), however, calculating 2-wise coverage is still time consuming (even though t is just 2). Especially, thousands of fitness evaluations are required in search-based test suite generation algorithms, leading to extremely long execution time. To alleviate this issue, *test suite diversity* or dissimilarity, which has been shown to be a scalable and flexible alternative to t-wise coverage [42], is adopted to define fitness functions. Motivated by our previous work [96], the following fitness function is used for large FMs:

$$f_{div}(TS) = \sum_{i=1}^{N} \rho(tc_i), \tag{3}$$

where

$$\rho(tc_i) = \frac{1}{k} \sum_{j=1}^{k} d(tc_i, tc_{ij}). \tag{4}$$

In Equation (4), tc_{ij} is the j-th nearest neighbor to tc_i in the test suite TS, and $d(tc_i, tc_{ij})$ denotes the distance between them. Although any set-based distances, such as Jaccard distance [42], can be applied here, we choose in this work Anti-dice distance [24], as given in Equation (5), due to its high performance observed in the context of SPL testing [96]:

$$d(tc_i, tc_{ij}) = 1 - \frac{|tc_i \cap tc_{ij}|}{2|tc_i \cup tc_{ij}| - |tc_i \cap tc_{ij}|}, \tag{5}$$

where $tc_i \cap tc_{ij}$ and $tc_i \cup tc_{ij}$ denote the intersection and union of tc_i and tc_{ij},[5] respectively, and $|\cdot|$ returns the cardinality of a set.

In fact, the fitness function given by Equation (3) is a generation of the *novelty score*, a key concept in the NS algorithm [26, 54], from a single test case to a test suite. According to Equation (4), the novelty score of a test case is defined as the average distance to its k nearest neighbors (k is a hyperparameter, often set to half of the test suite size [96]), and Equation (3) generalizes this score to a test suite by accumulating novelty scores over all members. It has been well demonstrated that the fitness function f_{div} has a significantly positive correlation with the t-wise coverage in most cases [96]. This is the main reason we choose f_{div} as a surrogate metric for the t-wise coverage. We must mention that f_{div} was originally defined in our previous work [96] and used to explore correlations between novelty score and t-wise coverage. In this work, we explicitly use it as an objective to guide the search for the test suite with the maximum diversity, and thus potentially gaining high t-wise coverage.

To sum up, the fitness in our proposed mathematical model is related to t-wise coverage, particularly pairwise coverage. For small FMs, t-wise coverage as given in Equation (1) is directly used as the fitness function. For large FMs, however, optimizing directly the t-wise coverage is quite time consuming and unscalable; we instead seek to optimize its surrogate metric—that is, test suite diversity defined by Equation (3).

3.4 Remarks on the Mathematical Model

To model SPL testing (or a general test task) as a QD problem, key issues to be addressed are defining the fitness function and the behavior space. The former is often related to test quality, whereas the latter is usually customized by software engineers based on their own preferences or testing scenarios. In this work, the behavior space considered is just one-dimensional, but this

[5] According to Definition 2.2, a test case is a set. Therefore, intersection and union of two test cases are meaningful.

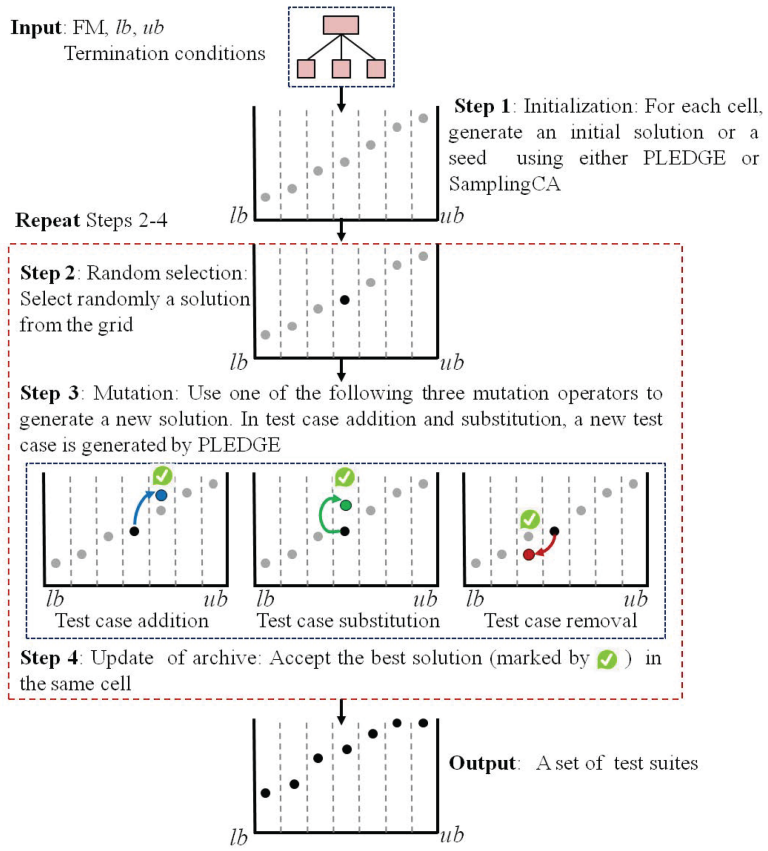


Fig. 3. Overview of MAP-Elites for automated test suite generation.

is already enough to show the benefits of QD optimization in comparison with single and multi-objective optimization. Definitely, if necessary, one can also consider higher-dimensional behavior spaces. This extension is explicitly mentioned as one of our future studies in Section 9.

4 MAP-ELITES FOR AUTOMATED TEST SUITE GENERATION

To solve the proposed QD model, we use one of the famous QD algorithms: MAP-Elites [18, 72]. It takes inspiration from evolutionary algorithms, and is characterized by a multi-dimensional archive, which physically stores the best-performing solution for each cell of the behavior space $\mathcal{B}$. Recall in Section 3.2 that $\mathcal{B}$ is often bounded and discretized into a grid with multiple cells. Figure 3 provides an overview of MAP-Elites for automated test suite generation in SPL testing: it takes an FM, lower and upper bounds of the grid (*lb* and *ub*), and termination conditions (maximum running time or number of iterations) as input, and outputs a set of test suites from which software testers can choose. The algorithm consists of the following four main steps. *Step 1* initializes the grid-based archive with either random solutions or seeds; *Step 2* selects a random solution from the grid. *Step 3* mutates the selected solution using either test case addition, test case substitution, or test case removal, and *Step 4* updates the grid accordingly (i.e., accepting the best solution in the same cell). Steps 2, 3, and 4 are repeated until the termination condition is met. More formally, Algorithm 2 gives pseudocode of MAP-Elites for automated test suite generation. In the following

ALGORITHM 2: MAP-Elites for automated test suite generation

```
   Input: FM, lb, ub
   Output: 𝒜 (archive)
1  𝒜 ← create_empty_archive(lb, ub) /*𝒜 is implemented by an array-based list*/;
2  M ← ub − lb + 1 /*number of cells*/;
3  for i ← 1 to M do
4  |  x ← a random solution or a seed with test suite size lb + (i − 1);
5  |  𝒜 ← add_to_archive(𝒜, x);
6  end
7  while termination condition is not met do
8  |  x ← selection(𝒜) ;
9  |  x′ ← mutation(x, FM) ;
10 |  𝒜 ← add_to_archive(𝒜, x′);
11 end
12 return 𝒜
```

subsections, we will present more details on the algorithm implementation, including initialization, selection/mutation, and update of the archive.

4.1 Initialization

According to line 1 of Algorithm 2, the initialization phase creates an empty archive $\mathcal{A}$ whose size is the same as the discretized behavior space. As discussed in Section 3.2, the behavior space in our case is a one-dimensional grid with $(ub-lb)/\delta$ cells ($\delta = 1$ is a constant). Therefore, $\mathcal{A}$ is initialized to a one-dimensional array with the same length, and each of its entries is set to *null*, indicating no solutions stored. In addition, according to lines 2 through 6, $M = (ub - lb + 1)$ solutions are initialized and potentially added into the archive (details are available in Section 4.3). Depending on whether or not seeds are implanted, we can consider the following two scenarios. The first one, referred to as `initialization_without_seeds`, randomly generates all the initial solutions using PLEDGE[6] [44], whereas the second one, referred to as `initialization_with_seeds`, generates $M-1$ solutions using PLEDGE [44], and an additional one (called a *seed*) using SamplingCA [67]. Note that SamplingCA returns a covering array (Definition 2.6), whereas PLEDGE aims at efficiently generating a test suite with diverse test cases, not necessarily a covering array.

4.2 Selection and Mutation Operators

At each iteration, as in traditional evolutionary algorithms, a solution $\boldsymbol{x}$ is selected from the current archive $\mathcal{A}$. Following the original MAP-Elites, a random selection is performed here. This means that each solution in $\mathcal{A}$ has an equal chance of being chosen. It should be noted that random selection has the merit of being highly efficient, and introducing no bias toward specific areas of the behavior space. For example, the selected solution in Figure 3 is marked as a dark solid circle (•).

Then, the selected solution $\boldsymbol{x}$ is copied and inputted into *mutation* to generate new solutions. We use three simple mutation operators from the work of Parejo et al. [77], and they are test case removal, test case substitution, and test case addition. In Step 3 of Figure 3, an illustration of the three operators is provided. Test case addition adds a random test case to the test suite, whereas the remaining two operators (i.e., test case removal and test case substitution) remove

[6]To generate a solution (i.e., a test suite), PLEDGE repeatedly requests to a modified version of SAT4J [42] to generate a test case.

ALGORITHM 3: $\boldsymbol{x}' \leftarrow mutation(\boldsymbol{x}, FM)$

```
   Input: x (a solution/test suite), FM
   Output: x′ (a mutated solution)
1  size ← the test suite size of x;
2  if size = lb then
3  |  r1 ← random(0, 1);
4  |  if r1 < 2/3 then
5  |  |  x′ ← TestCase_Substitution(x, FM);
6  |  else
7  |  |  x′ ← TestCase_Addition(x, FM);
8  |  end
9  else if size = ub then
10 |  r2 ← random(0, 1);
11 |  if r2 < 2/3 then
12 |  |  x′ ← TestCase_Substitution(x, FM);
13 |  else
14 |  |  x′ ← TestCase_Removal(x);
15 |  end
16 else
17 |  r3 ← random(0, 1);
18 |  if r3 < 1/3 then
19 |  |  x′ ← TestCase_Substitution(x, FM);
20 |  else if 1/3 ≤ r3 < 2/3 then
21 |  |  x′ ← TestCase_Removal(x);
22 |  else
23 |  |  x′ ← TestCase_Addition(x, FM);
24 |  end
25 end
26 return x′
```

and substitute a test case at a randomly chosen place of the test suite, respectively. Clearly, a test suite can be improved either by itself via test case substitution, or by leveraging adjacent test suites via test case removal and test case addition. Notice that a new test case is required in both test case substitution and test case addition, and it is generated by applying also the tool PLEDGE [44]. For instance, consider the illustrative example in Section 2.1, and suppose that the selected solution is a test suite with five test cases, $\boldsymbol{x} = \{tc_1, tc_3, tc_4, tc_7, tc_8\}$. By adding a new test case tc_5 and removing the existing test case tc_8, new test suites $\boldsymbol{x}'_1 = \{tc_1, tc_3, tc_4, tc_5, tc_7, tc_8\}$ and $\boldsymbol{x}'_2 = \{tc_1, tc_3, tc_4, tc_7\}$ are generated, respectively. Similarly, substituting tc_3 with tc_2 also leads to a new solution $\boldsymbol{x}'_3 = \{tc_1, tc_2, tc_4, tc_7, tc_8\}$. Different from $\boldsymbol{x}'_1$ and $\boldsymbol{x}'_2$, the solution $\boldsymbol{x}'_3$ lies in the same cell as the original solution $\boldsymbol{x}$.

Algorithm 3 outlines the procedure of *mutation*. Given that the generation of a new test case in both *TestCase_Substitution* and *TestCase_Addition* needs to specify the FM, both of them have two parameters: $\boldsymbol{x}$ and FM. As can be seen, combinations of the used mutation operators are determined by the size of the test suite $\boldsymbol{x}$, denoted by *size* (see line 1 of Algorithm 3). Specifically, if $\boldsymbol{x}$ is a test suite with *size* being the lower bound lb, then test case removal is impossible. Thus, according to lines 4 through 8, only test case substitution and test case addition are used, with probability 2/3 and 1/3, respectively. Analogously, if *size* equals to the upper bound ub, then only test case substitution and test case removal are adopted, as shown in lines 11 through 15. In a general

ALGORITHM 4: $\mathcal{A} \leftarrow add\_to\_archive(\mathcal{A}, \boldsymbol{x}')$

Input: $\boldsymbol{x}'$ (a new solution), $\mathcal{A}$ (current archive)
Output: $\mathcal{A}$ (updated archive)

```
1 {f, b} ← evaluate(x') ;
2 c ← get_cell_index(b) ;
3 if A(c) = null or A(c).f < f then
4 |   A(c) ← {x', f};
5 end
6 return A
```

case in which $lb < size < ub$, lines 18 through 24 indicate that the three mutation operators are simultaneously used, with probability 1/3 for each of them. We should mention that the weight for test case substitution is 2/3, rather than 1/3, when *size* is equal to lb or ub. This is to ensure that the trials to improve each cell (via either substitution, addition, or removal) are equal on average. Assume that each cell is selected 2,000 times in the whole search process. When the cell lb is selected, the corresponding solution $\boldsymbol{x}$ can be possibly improved by either test case substitution operated on $\boldsymbol{x}$ itself or test case removal operated on the solution in the cell $lb + 1$. If test case substitution is invoked with probability 1/3 in this case, then the cell lb is improved with only around $(1/3 + 1/3) \times 2{,}000$ trials, less than 2,000 trails. However, this issue will be fixed by increasing the probability to 2/3.

As discussed in the work of Chatzilygeroudis et al. [14], the key to the success of QD algorithms (including MAP-Elites) is the information sharing between the optimization subproblems. Thanks to test case removal and test case addition, the mutation implemented by Algorithm 3 is able to exchange information among adjacent cells. In Section 6.1.3, we will show that, for each cell, successful updates brought by test case removal/addition are generally more than those brought by test case substitution, indicating the usefulness and indispensability of the two mutation operators. More detailed discussions will be given later in Section 6.1.3.

4.3 Update of Archive

The update of the archive is straightforward, with details given in Algorithm 4. As in the original MAP-Elites, this update procedure aims at filling every cell of the grid with the best possible solution. More specifically, as shown in line 1 of Algorithm 4, the fitness function f and the behavior descriptor $\boldsymbol{b}$ of the solution $\boldsymbol{x}'$ are worked out by calling the *evaluate* function. Then, we can get the index c of the cell where $\boldsymbol{x}'$ is located based on the behavior descriptor $\boldsymbol{b}$. Note that since only one behavior is considered in our case, $\boldsymbol{b}$ degenerates into a scalar value, representing the test suite size. Therefore, the index c can be simply computed as $c = \boldsymbol{b} - lb$. According to lines 3 through 5 of Algorithm 4, if the cell with index c is empty (*null*) or contains a solution performing worse than $\boldsymbol{x}'$ regarding the fitness value, then $\boldsymbol{x}'$ associated with f occupies that cell. For example, suppose that the original solution is $\boldsymbol{x} = \{tc_1, tc_3, tc_4, tc_7, tc_8\}$, and the new solution is $\boldsymbol{x}' = \{tc_1, tc_2, tc_4, tc_7, tc_8\}$. We just need to compare $\boldsymbol{x}'$ with $\boldsymbol{x}$ regarding the fitness value (2-wise coverage or test suite diversity) and keep the one that performs better.

4.4 Remarks on the Algorithm Design

To summarize, applying MAP-Elites in the context of this work requests software engineers to specify a region of test suite size that they are interested in. The intuition is that in practice, it may be difficult for software engineers to identify a test suite size matching exactly their test budgets. Instead, it would be much easier for them to roughly specify a range. MAP-Elites aims at generating

the best possible test suite for each of the specified sizes, thus providing multiple alternatives for software engineers to choose from.

To make MAP-Elites effective, it is of great importance to design variation (e.g., mutation) operators that enable information sharing between adjacent cells. The mutation given in Algorithm 3 is customized for such a purpose. In the mutation, PLEDGE is adopted to generate new test cases. However, it does not take into account what t-sets are already covered in the test suite. Therefore, when t-wise coverage is chosen as the fitness function, using only PLEDGE will lead to ineffectiveness. However, this issue can be remedied by implanting seeds (see Section 4.1). According to other works [45, 60], seeding is a widely used strategy in search-based testing for SPLs, and does have a positive impact on the performance of the search algorithm. Because "seeds" include some desirable building blocks (i.e., t-sets) of the optimal solutions, they can help the algorithm in accelerating its search. As will be shown in Section 6.3, MAP-Elites with the seeding strategy can reduce the size of the covering array returned by state-of-the-art t-wise testing tools in at least one run on almost all the FMs under study. When the fitness function is test suite diversity, however, using PLEDGE could be effective because its goal is to generate diverse test cases [42].

We should mention that this work sticks mostly to the original implementation of MAP-Elites because our primary goal is to show the feasibility and effectiveness of this kind of algorithms. As will be explicitly mentioned in Section 9, advanced MAP-Elites (or broadly QD algorithms) can also be applied in the future.

5 EXPERIMENTAL DESIGN

The proposed approach, SPL test suite generation based on MAP-Elites, is evaluated through experiments in which it is compared with several alternatives. The performed experiments aim at addressing the following four **Research Questions (RQs)**:

- *RQ1*: How does MAP-Elites perform, in terms of t-wise coverage and test suite diversity, when compared with the vanilla algorithm where each QD subproblem is independently optimized by a single-objective approach?
- *RQ2*: How does MAP-Elites perform, in terms of t-wise coverage, when compared with NSGA-II, one of the most famous MOEAs?
- *RQ3*: How does MAP-Elites perform in comparison with state-of-the-art testing tools?
- *RQ4*: How does MAP-Elites perform, in terms of fault detection capacity, when compared with the NS-based test suite generation algorithm [96] that completely ignores objectives?

To address RQ1, we compare MAP-Elites with the vanilla algorithm where each QD subproblem is independently optimized by a single-objective approach. This vanilla algorithm is named MI-GA (*M*ultiple *I*ndependent run of *G*enetic *A*lgorithms), where a random solution (i.e., test suite) is initialized for each QD subproblem, then independently improved via mutation operation that introduces only test case substitution. The essential difference between the two algorithms is that MI-GA disables information sharing between QD subproblems. As discussed in the work of Chatzilygeroudis et al. [14], the central hypothesis of QD algorithms is that, due to the benefits brought by information sharing, solving a set of problems together is likely to be better/faster than they are solved independently. RQ1 aims at demonstrating that this hypothesis holds in our context by comparing MAP-Elites with MI-GA in terms of both t-wise coverage and test suite diversity. Addressing RQ1 enables us to demonstrate suitability and usefulness of MAP-Elites as a tool for generating (multiple) test suites at a time.

To address RQ2, we independently run both MAP-Elites and NSGA-II, and compare the t-wise coverage results. Recall in Section 3 that the test suite generation is modeled as a QD problem where pairwise coverage and test suite size are used as the fitness and measure functions,

respectively. In fact, they have also been used as the two objectives in the bi-objective model built previously [59–61]. Lopez-Herrejon et al. [61] have applied classical MOEAs (e.g., NSGA-II [23]) to solve the problem. Since NSGA-II and MAP-Elites share the same ultimate goal (i.e., providing the best solution for each possible test suite size, offering a set of solutions from which testers can choose [61]), it is meaningful to investigate how MAP-Elites performs in comparison with NSGA-II. This is what we will answer in RQ2.

To address RQ3, we compare MAP-Elites with three existing t-wise testing tools: YASA [50], IncLing [1], and SamplingCA [67]. Although these testing tools generate only a single test suite each time, multiple test suites can be trivially derived from it by simply removing one by one the last test case. In this article, we call them *derived test suites*. For example, assume that the test suite generated by such tools has eight test cases, removing the last test case will create a derived test suite with size 7, and removing again the last one will generate a derived test suite with size 6, and so on. Therefore, comparisons between MAP-Elites and these tools are meaningful. In fact, YASA [50] and IncLing [1] are relatively new among all the five t-wise testing tools evaluated in a recent comparative study [32]. SamplingCA [67], which was proposed in 2022, is a sampling-based approach for the covering array generation problem. Experimental results on 125 public SPLs demonstrate that SamplingCA can generate much smaller pairwise covering arrays, and also runs one to two orders of magnitude faster than state-of-the-art competitors [67]. In addition to the preceding three tools, MAP-Elites is also compared with GrES [45], which is a search-based multi-objective approach where test suite size is considered among the objectives to minimize. Note that minimizing test suites is useful when tests need to be looked at manually or executing test cases is time consuming in practical case studies [69]. One of the main characteristics of GrES is that it optimizes first on pairwise coverage and then on the other objectives (e.g., test suite size). Starting from the seed generated by any existing t-wise testing tool, both MAP-Elites and GrES have the potential ability to find a smaller covering array than the seed. We will compare MAP-Elites against GrES regarding this ability in RQ3.

Finally, RQ4 amounts to evaluating the fault detection capacity of the algorithms. To address RQ4, we compare MAP-Elites with a highly related algorithm called *NS* [96] in terms of how many faults they disclose. NS was originally proposed as a pure diversity algorithm [54], and was recently adapted to generate an SPL test suite [96]. Although both MAP-Elites and NS belong to QD algorithms, and both of them are applied to SPL test suite generation, they have the following major differences. First, NS searches for only one test suite, whereas MAP-Elites searches for multiple ones at a time. Second, the search in NS ignores objectives, but is driven by encouraging solutions contributing largely to the novelty score (measuring diversity). In contrast, MAP-Elites explicitly treats test suite diversity or t-wise coverage as an objective and optimizes it by emphasizing solutions with high fitness values.

5.1 Subject Product Lines

In the experiments, we use a set of 105 FMs whose characteristics are depicted in Table 2. For each FM, Table 2 presents its name, the number of features, the number of CNF constraints, and the number of valid 2-sets (only applicable to FMs with fewer than 100 features). Note that these FMs were mainly collected from four related papers [7, 42, 45, 96], and they have been widely used as benchmarks to assess SPL testing techniques [7, 42, 45, 49, 60, 66, 77, 78, 96].

For these FMs, 64 of them are realistic, whereas the remaining 41 FMs are artificially generated. By *realistic*, we mean that these models represent real-world SPLs whose code is publicly available or accessible under request to the respective authors. There are two groups of artificial FMs that are generated by using different tools. Specifically, the FMs named "`n*Model*`" were originally generated by Hierons et al. [45] using the tool BeTTy [86], whereas those with the name prefix

Table 2. FMs Used in the Experimental Study

FM	#Features	#Constraints	#2-sets	FM	#Features	#Constraints
ZipMe	8	9	85	n100Model1	116	205
BerkeleyDBFootprint	9	9	128	n100Model2	120	220
Apache	10	11	145	n100Model3	116	218
argo-uml-spl	11	13	163	n100Model4	119	206
LLVM	12	13	221	n100Model5	120	214
PKJab	12	16	177	n100Model6	117	212
Curl	14	19	307	n100Model7	120	208
Wget	17	22	475	n100Model8	122	235
x264	17	26	469	n100Model9	121	218
BerkeleyDBC	18	29	529	n100Model10	115	202
gpl	18	40	418	Printers	172	310
BerkeleyDBMemory	19	37	625	fiasco_17_10	234	1,178
fame_dbms_fm	21	38	589	uClibc-ng_1_0_29	269	1,403
DesktopSearcher	22	38	674	E-shop	290	426
CounterStrikeSimpleFM	24	35	833	toybox	544	1,020
BerkeleyDBPerformance	27	45	1,063	axTLS	684	2,155
LinkedList	27	48	980	busybox_1_28_0	998	962
SensorNetwork	27	43	1,215	SPLOT-FM-1000-1	1,000	1,875
HiPAcc	31	104	1,712	SPLOT-FM-1000-2	1,000	1,927
SPLSSimuelESPnP	32	54	1,448	SPLOT-FM-1000-3	1,000	1,933
TankWar	37	59	2,157	SPLOT-FM-1000-4	1,000	1,807
JavaGC	39	105	2,399	SPLOT-FM-1000-5	1,000	1,889
Polly	40	100	2,402	SPLOT-FM-1000-6	1,000	1,814
DSSample	41	201	2,592	SPLOT-FM-1000-7	1,000	1,874
VP9	42	104	2,695	SPLOT-FM-1000-8	1,000	1,897
WebPortal	43	68	3,196	SPLOT-FM-1000-9	1,000	1,788
JHipster	45	104	3,151	SPLOT-FM-1000-10	1,000	1,935
Drupal	48	79	3,751	mpc50	1,213	3,728
SmartHomev2.2	60	82	6,189	ref4955	1,218	3,099
VideoPlayer	71	99	7,528	linux	1,232	3,154
Amazon	79	250	10,555	csb281	1,233	3,114
ModelTransformation	88	151	13,139	ecos-icse11	1,244	3,146
CocheEcologico	94	191	11,075	ebsa285	1,245	3,832
n30Model1	35	57	1,897	vrc4373	1,247	3,104
n30Model2	35	64	1,808	pati	1,248	3,266
n30Model3	34	61	1,724	dreamcast	1,252	3,168
n30Model4	34	62	1,567	pc_i82544	1,259	3,179
n30Model5	34	59	1,658	XSEngine	1,260	3,803
n30Model6	36	70	1,549	refidt334	1,263	3,140
n30Model7	34	64	1,662	ocelot	1,266	3,141
n30Model8	35	63	1,784	integrator_arm9	1,267	50,606
n30Model9	33	51	1,670	olpcl2294	1,273	3,878
n30Model10	36	60	2,063	olpce2294	1,274	3,881
n50Model1	56	93	3,530	phycore	1,274	3,852
n50Model2	58	95	5,380	hs7729pci	1,298	49,911
n50Model3	58	95	5,380	freebsd-icse11	1,396	62,183
n50Model4	56	94	5,408	uClinux	1,850	2,468
n50Model5	58	99	5,606	Automotive01	2,513	10,311
n50Model6	58	103	4,918	SPLOT-FM-5000	5,000	9,419
n50Model7	58	107	5,505	busybox-1.18.0	6,796	17,836
n50Model8	56	99	4,300	2.6.28.6-icse11	6,888	343,944
n50Model9	59	101	5,182	–	–	–
n50Model10	63	110	6,050	–	–	–

"SPLOT-FM" are generated using **Software Product Line Online Tools (SPLOT)** [71]. Both tools generate FMs in SXFM (Simple XML Feature Model) [71] format, and we transform them into DIMACS format using the export functionality of FeatureIDE [90].

According to Table 2, FMs used in this study are representative regarding the number of features and constraints. The smallest model is ZipMe, with 8 features and 9 constraints, whereas the largest one is 2.6.28.6-icse11 with 6,888 features and 343,944 constraints. FMs on the left of the table are relatively small, whereas those on the right are large. Taking into account both small and large FMs enables us to investigate the scalability of the algorithms.

5.2 Performance Metrics

For RQ1 through RQ3, we use the QD-Score [81] as the performance metric. The QD-Score was proposed to evaluate different QD algorithms, and it was defined as the total fitness values across all filled cells within the returned archive [81]. Formally,

$$QD\text{-}Score(\mathcal{A}) = \sum_{c} f\left(\mathcal{A}(c)\right), \tag{6}$$

where $\mathcal{A}$ is the archive returned by an algorithm, c denotes index of the cell, and f is the fitness value of a solution. In this work, the fitness function can be either t-wise coverage defined by Equation (1) or test suite diversity defined by Equation (3).

As mentioned before, we focus mainly on 2-wise coverage. For its calculation, all valid 2-sets are required. Proposed by Henard et al. [42], and adopted in our previous work [96], the following method is used to compute all valid 2-sets of an FM. First, compute all the possible 2-wise feature combinations from the set $\{f_1, f_2, \ldots, f_n, -f_1, -f_2, \ldots, -f_n\}$.[7] Then, check the validity of the combinations using SAT solvers (e.g., SAT4J [10]) and remove those that are invalid. Table 2 gives the number of valid 2-sets for small FMs. For large ones, we use test suite diversity as the fitness, which can be calculated following exactly the descriptions in Section 3.3.

RQ4 examines the fault detection capability of the algorithms. As in prior work [84, 96], we adopt fault detection rate as the performance metric, which is defined as the ratio of the number of detected faults to that of all faults. A fault in this context means a feature combination that triggers bugs. Due to lack of source code and unit tests for the FMs in Table 2, following the common practice in other works [2, 6, 29, 85], we use simulated faults that are generated by the following procedure. First, set t to a random integer between 1 and 6 (inclusive). Then, pick t features in a random manner and make each of them either selected or deselected both with probability 0.5. Third, check validity of this feature combination via SAT solvers. Note that only valid feature combinations are accepted as faults. Repeat the preceding steps until the number of faults reaches the desired value. As suggested in the work of Al-Hajjaji et al. [2], the number of simulated faults for each FM is set to $n/10$, with n being the number of features. Following Sánchez et al. [85], we assume a simple oracle to detect faults: a fault is detected if it is contained in any test case of a test suite. It must be mentioned that, to mitigate random biases, 100 sets of simulated faults are generated for each FM, and that the reported fault detection rate is averaged over all 100 fault sets.

5.3 Implementation Details and Parameter Settings

In the experiments for RQ1, the upper bound ub for the grid can be set to the size of the test suite generated by any t-wise testing algorithm. We here choose SamplingCA[8] [67] due to its good scalability and high effectiveness. Regarding the lower bound lb, it is set to $\max\{2, ub - 99\}$. For

[7] A feature has two states, being either selected or deselected.

[8] The code of SamplingCA is obtained from https://github.com/chuanluocs/SamplingCA

example, for a run on a particular FM, SamplingCA generates a test suite with size 10, then ub and lb in MAP-Elites are set to 10 and 2, respectively. This is to say, we will search for $ub - lb + 1$ (i.e., $10 - 2 + 1 = 9$) best test suites at a time, with their sizes being 2 at least and 10 at most. On some models (e.g, `Amazon`), SamplingCA generates a test suite with size 264, then $ub = 264$ and $lb = \max\{2, ub - 99\} = 165$. The preceding setting of lb enables that the number of test suites to generate is at most 100. However, this number can be flexibly determined by software engineers based on their own preferences or search budgets. The termination criterion in both MAP-Elites and MI-GA is a pre-defined number of fitness **Function Evaluations (FEs)**, which is set to $500 \times (ub - lb + 1)$, where $(ub - lb + 1)$ is the number of test suites, or archive size. On average, each QD subproblem is improved with 500 FEs. In addition, we run experiments where the number of test suites is fixed to 6 (by simply setting lb to $ub - 5$), and the number of FEs is increased to 2,000 for each QD subproblem. Recall in Section 4.1 that depending on whether or not seeds are implanted, there exist two scenarios, `initialization_with_seeds` (or `with seeds` for simplicity) and `initialization_without_seeds` (or `without seeds` for simplicity), for each of which experiments are performed and results are compared. Distinguishing the two scenarios seeks to make clear whether or not our conclusions to be drawn are sensitive to seeding.

In the experiments for RQ2, we compare MAP-Elites with NSGA-II, one of the MOEAs examined by Lopez-Herrejon et al. [60] in the context of multi-objective SPL testing. For each FM and each run, the population size in NSGA-II is the same as the archive size in MAP-Elites, which is the same setting as in the experiments for RQ1 and can be as large as 100. Moreover, NSGA-II adopts the same mutation operator as given in Algorithm 3 to generate new solutions. Being in line with RQ1, the number of FEs in both algorithms, without using seeds, is set to 500 for each subproblem. To fairly compare the solution sets returned by the two algorithms, we put the solutions of NSGA-II into a QD-type archive. Specially, each solution is placed in the corresponding slot based on its test suite size. Note that if multiple solutions occupy the same slot, then only the best one is kept. We should also mention that if the pairwise coverage of the solution TS_i reaches the maximum value 100%, then all slots after TS_i's are automatically filled with the pair $(TS_i, 100\%)$. The rationale is that any larger test suite containing TS_i must have 100% pairwise coverage. It is possible that there are empty cells in the archive of NSGA-II (to be shown and explained later in Section 6.2). Following the common tradition in QD-related studies [81], we only add pairwise coverage in non-empty cells when calculating the QD-Score. Following Lopez-Herrejon et al. [60], NSGA-II is implemented by using the code provided in jMetal.[9]

In the experiments for RQ3, MAP-Elites is first compared with YASA, IncLing,[10] and SamplingCA using the same time budget. For each model and each run of the preceding three tools, we record the consumed time (in seconds) and the returned test suite. Then, as done in RQ1, ub in MAP-Elites is set to the size of the returned test suite, and lb is set to $\max\{2, ub - 99\}$. In MAP-Elites, no seeds are used and the fitness function is the test suite diversity defined by Equation (3). Note that using no seeds aims at making fair comparisons. In addition, we use test suite diversity, instead of 2-wise coverage, because the former is computationally cheaper. Then, MAP-Elites is compared with GrES[11] in the second experiment where both algorithms are used to find possibly a test suite with full 2-wise coverage but smaller than the seed generated by YASA, IncLing, and SamplingCA. Therefore, starting from the population initialized by the derived test suites of these testing tools, we run MAP-Elites using 2-wise coverage as the fitness function and terminate it after $500 \times (ub - lb + 1)$ evaluations. Here ub is set to the size of the seed, and lb is set to $ub - 5$. These

[9] https://github.com/jMetal/jMetal

[10] Both YASA and IncLing are implemented in FeatureIDE [90], available at https://github.com/FeatureIDE/FeatureIDE

[11] The code of GrES is obtained from https://drive.google.com/open?id=1xumU6qxBesloq69jOPMbprOaiaOKDq82

settings focus the search on test suites slightly smaller than the seed. Indeed, it is rarely possible to find a covering array significantly smaller than the seed generated by existing t-wise testing tools. For GrES, we choose two objectives: pairwise coverage and test suite size. Even though nine objectives were considered in the original study of GrES, as explicitly mentioned by its authors, one can choose to use a subset of these. Moreover, only three objectives (i.e., pairwise coverage, test suite size, and test suite cost) are relevant to test case selection (the scope of this work).[12] Because test suite cost is not considered in the context of this work, we abandon it and consider only pairwise coverage and test suite size. Note that in the experiments for RQ2, the two objectives are also used in NSGA-II. Different from NSGA-II which treats them equally, GrES views pairwise coverage more important than test suite size. This is a core characteristic of GrES. Hence, we use two simple rules to compare two test suites in the (environmental) selection process of GrES: (1) preferring the test suite with higher pairwise coverage and (2) preferring the smaller test suite when their pairwise coverage is equal. GrES starts the search from the derived test suite with size lb, using the same mutation operators, the same number of FEs as in MAP-Elites.

In the experiments for RQ4, MAP-Elites (where seed is not adopted and test suite diversity is used as the fitness function) is compared with NS[13] [96], which returns a single test suite instead of multiple ones. For NS, we consider two values (i.e., 50 and 100) for the test suite size and three values (i.e., 500, 1,000, and 2,000) for the number of FEs. In MAP-Elites, ub and lb are set according to the test suite size in NS. More specifically, when the test suite size is 100 in NS, ub and lb are set to 101 and 99, respectively. Thus, the search is focused on the best test suite with around 100 test cases. Similarly, $ub = 51$ and $lb = 49$ when the test suite size in NS is 50. Since MAP-Elites searches three test suites at a time, the search budget is tripled accordingly. Notice that, to be fair, we compare the test suite returned by NS with the same-sized one returned by MAP-Elites in terms of the fault detection rate.

In RQ1 and RQ2, we compare MAP-Elites with both single- and multi-objective approaches, and want to know whether or not our conclusions are sensitive to the seeding strategy and the fitness functions used. This leads to four combinations: (with seeds, 2-wise coverage), (with seeds, test suite diversity), (without seeds, 2-wise coverage), and (without seeds, test suite diversity). However, for large FMs, both generating seeds and optimizing the 2-wise coverage are (extremely) expensive, leading to the difficulty in conducting experiments for all preceding four combinations. In RQ3, we also need to generate seeds and optimize the 2-wise coverage. Therefore, we use only small FMs for answering RQ1 through RQ3. For answering RQ4, we turn to large FMs because seed is not adopted and test suite diversity (which scales well) is used as the fitness function.

Due to the stochastic nature, the algorithms are independently executed 30 times for each experiment to reduce randomness. All the algorithms, except for GrES which was written in C, are implemented using Java, and executed in a workstation equipped with an Intel Core i7-7700 CPU@3.60 GHz and 8 GB of RAM, running Window 10 Enterprise Edition. All source codes of our algorithm, along with the used FMs, are available on GitHub.[14]

5.4 Statistical Test Tools

To have sound comparisons, statistical test tools are required. First, the Mann-Whitney U test with a 0.05 significance level is adopted to examine the significance of the difference between the results obtained by MAP-Elites and its competitors. For simplicity, statistical test results are represented by three symbols, •, ∗, and ∘, indicating that MAP-Elites performs significantly better

[12] All the remaining six objectives are related to test case prioritization (beyond the scope of this work).
[13] The code of NS is downloaded from https://github.com/gzhuxiangyi/TSE_NS
[14] https://github.com/gzhuxiangyi/SPLTestingMAP

than, equivalently to, and significantly worse than the competitors, respectively. Second, following Arcuri and Briand [3], the Vargha and Delaney's $\widehat{A}_{12}$ statistic[15] [91] is utilized to evaluate effect sizes—that is, determine which algorithm leads to better results and to what extent [3, 4]. In addition, $\widehat{A}_{12}$ test results are represented by three symbols, ↑, ∼, and ↓, indicating that MAP-Elites provides better, equal, and worse results than the compared algorithms, respectively. Moreover, according to Vargha and Delaney [91], differences between algorithms can be assessed as large (l), medium (m), small (s), and negligible (n). For example, the symbol combination (↑, l) means that MAP-Elites has a large improvement over its competitor, whereas (↓, l) indicates the opposite case (i.e., a large degeneration). Note that the symbol ∼ is always accompanied with n, meaning negligible differences between the two algorithms. In brief, the $\widehat{A}_{12}$ statistic provides a qualitative way of evaluating the magnitude that MAP-Elites performs in comparison with relevant algorithms.

6 RESULTS AND DISCUSSION

In this section, we outline results of the performed experiments. In the tables of this section, the best values of performance metrics are highlighted in boldface type. Moreover, Mann-Whitney U test results and $\widehat{A}_{12}$ statistics are marked by the corresponding symbols as explained in Section 5.4.

6.1 RQ1: Comparison with MI-GA

In this section, we compare MAP-Elites with MI-GA on small FMs given in Table 2, using different fitness functions and different initialization strategies (seeded or not seeded). We report the results and explain why MAP-Elites outperforms MI-GA. At the end of this section, a summary of this RQ is provided.

6.1.1 t-Wise Coverage as the Fitness Function. Table 3 gives the QD-Score for the 2-wise coverage obtained by MAP-Elites and MI-GA, in each of which the fitness function is set to the 2-wise coverage given by Equation (1). As explained before, MAP-Elites solves all the QD subproblems collaboratively, whereas MI-GA solves them independently. Concerning the QD-Score (averaged over 30 runs), as can be seen from Table 3, MAP-Elites outperforms MI-GA on all the FMs, regardless of whether or not seeds are implanted. Moreover, Figure 4(a) summarizes the Mann-Whitney U test results, which indicate that, no matter whether the initial population is seeded or not, MAP-Elites performs significantly better than and equivalently to MI-GA on 77% and 23% of all the FMs, respectively. It is worth emphasizing that there are no FMs on which MAP-Elites is significantly inferior to MI-GA. Regarding the $\widehat{A}_{12}$ statistic, as found from Figure 4(b), MAP-Elites improves MI-GA on 100% of all the FMs, with either large (l), medium (m), or small (s) differences. In particular, large and medium effect sizes in favor of MAP-Elites are observed in 68% and 70% of cases for `With seeds` and `Without seeds`, respectively. Again, there are no cases in which MI-GA is favored. In Figure 5, we show the fitness value (2-wise coverage) for each of the test suites returned by MAP-Elites (`With seeds`) and MI-GA (`With seeds`) in a particular run in which the QD-Score is closest to the mean over all 30 runs. Note that FMs chosen in Figure 5 are representative regarding the number of features, representing either realistic or artificial SPLs. As can be seen, for each FM and each of the test suite returned in the run, the 2-wise coverage of MAP-Elites is generally higher than (or at least equal to) that of MI-GA. The preceding numerical results and visual comparisons clearly emphasize the superiority of MAP-Elites over MI-GA when 2-wise coverage is directly used as the fitness function.

Regarding the time cost, according to Figure 6, MAP-Elites runs fast on most of the FMs but slow on `Amazon`, `CocheEcologico`, `ModelTransformation`, and `DSSample`. According to Table 2,

[15]The $\widehat{A}_{12}$ statistic is performed using the *effsize* package in the R platform.

Table 3. QD-Score for 2-Wise Coverage Obtained by MAP-Elites and MI-GA Where the Fitness Function is 2-Wise Coverage

	With seeds				Without seeds			
	MAP-Elites	MI-GA	$\widehat{A}_{12}$		MAP-Elites	MI-GA	$\widehat{A}_{12}$	
ZipMe	**6.318e + 02**	6.294e + 02●	0.676	↑ *m*	**6.318e + 02**	6.294e + 02●	0.674	↑ *m*
BerkeleyDBFootprint	**6.063e + 02**	6.055e + 02∗	0.583	↑ *s*	**6.070e + 02**	6.063e + 02∗	0.618	↑ *s*
Apache	**6.179e + 02**	6.172e + 02∗	0.633	↑ *s*	**6.179e + 02**	6.166e + 02●	0.726	↑ *m*
argo-uml-spl	**6.273e + 02**	6.248e + 02●	0.736	↑ *m*	**6.270e + 02**	6.258e + 02●	0.652	↑ *s*
LLVM	**7.086e + 02**	7.072e + 02∗	0.600	↑ *s*	**7.095e + 02**	7.072e + 02∗	0.618	↑ *s*
PKJab	**6.446e + 02**	6.429e + 02●	0.674	↑ *m*	**6.444e + 02**	6.435e + 02●	0.698	↑ *m*
Curl	**9.818e + 02**	9.774e + 02●	0.835	↑ *l*	**9.824e + 02**	9.779e + 02●	0.818	↑ *l*
Wget	**9.758e + 02**	9.707e + 02●	0.717	↑ *m*	**9.757e + 02**	9.703e + 02●	0.719	↑ *m*
x264	**1.351e + 03**	1.344e + 03●	0.788	↑ *l*	**1.351e + 03**	1.343e + 03●	0.782	↑ *l*
BerkeleyDBC	**1.727e + 03**	1.715e + 03●	0.861	↑ *l*	**1.724e + 03**	1.715e + 03●	0.861	↑ *l*
gpl	**1.116e + 03**	1.107e + 03●	0.701	↑ *m*	**1.114e + 03**	1.106e + 03●	0.682	↑ *m*
BerkeleyDBMemory	**2.655e + 03**	2.637e + 03●	0.821	↑ *l*	**2.653e + 03**	2.637e + 03●	0.821	↑ *l*
fame_dbms_fm	**1.200e + 03**	1.196e + 03●	0.701	↑ *m*	**1.200e + 03**	1.196e + 03●	0.693	↑ *m*
DesktopSearcher	**8.220e + 02**	8.200e + 02●	0.664	↑ *s*	**8.228e + 02**	8.201e + 02●	0.707	↑ *m*
CounterStrikeSimpleFM	**9.103e + 02**	9.010e + 02●	0.704	↑ *m*	**9.076e + 02**	9.022e + 02●	0.689	↑ *m*
BerkeleyDBPerformance	**9.138e + 02**	9.100e + 02●	0.662	↑ *s*	**9.148e + 02**	9.099e + 02●	0.650	↑ *s*
LinkedList	**1.392e + 03**	1.386e + 03●	0.738	↑ *l*	**1.389e + 03**	1.385e + 03●	0.731	↑ *m*
SensorNetwork	**9.914e + 02**	9.768e + 02●	0.674	↑ *m*	**9.879e + 02**	9.768e + 02●	0.673	↑ *m*
HiPAcc	**3.072e + 03**	3.028e + 03●	0.694	↑ *m*	**3.057e + 03**	3.026e + 03●	0.694	↑ *m*
SPLSSimuelESPnP	**1.008e + 03**	1.004e + 03●	0.727	↑ *m*	**1.007e + 03**	1.003e + 03●	0.722	↑ *m*
TankWar	**1.285e + 03**	1.272e + 03●	0.683	↑ *m*	**1.281e + 03**	1.272e + 03●	0.683	↑ *m*
JavaGC	**4.112e + 03**	4.083e + 03∗	0.591	↑ *s*	**4.103e + 03**	4.080e + 03∗	0.591	↑ *s*
Polly	**3.077e + 03**	3.058e + 03∗	0.637	↑ *s*	**3.072e + 03**	3.058e + 03∗	0.637	↑ *s*
DSSample	**8.848e + 03**	8.781e + 03●	1.000	↑ *l*	**8.830e + 03**	8.781e + 03●	1.000	↑ *l*
VP9	**3.025e + 03**	3.006e + 03∗	0.617	↑ *s*	**3.022e + 03**	3.006e + 03∗	0.617	↑ *s*
WebPortal	**1.662e + 03**	1.634e + 03●	0.696	↑ *m*	**1.651e + 03**	1.635e + 03●	0.696	↑ *m*
JHipster	**3.484e + 03**	3.462e + 03●	0.670	↑ *m*	**3.478e + 03**	3.462e + 03●	0.670	↑ *m*
Drupal	**1.281e + 03**	1.261e + 03●	0.751	↑ *l*	**1.275e + 03**	1.260e + 03●	0.751	↑ *l*
SmartHomev2.2	**1.473e + 03**	1.446e + 03●	0.733	↑ *m*	**1.462e + 03**	1.446e + 03●	0.733	↑ *m*
VideoPlayer	**1.291e + 03**	1.280e + 03●	0.737	↑ *m*	**1.286e + 03**	1.279e + 03●	0.737	↑ *m*
Amazon	**9.885e + 03**	9.772e + 03●	1.000	↑ *l*	**9.813e + 03**	9.770e + 03●	1.000	↑ *l*
ModelTransformation	**2.665e + 03**	2.644e + 03●	0.672	↑ *m*	**2.659e + 03**	2.645e + 03●	0.672	↑ *m*
CocheEcologico	**8.744e + 03**	8.722e + 03∗	0.642	↑ *s*	**8.741e + 03**	8.721e + 03∗	0.642	↑ *s*
n30Model1	**1.105e + 03**	1.100e + 03●	0.738	↑ *l*	**1.105e + 03**	1.100e + 03●	0.717	↑ *m*
n30Model2	**1.308e + 03**	1.301e + 03●	0.663	↑ *s*	**1.306e + 03**	1.301e + 03∗	0.644	↑ *s*
n30Model3	**1.200e + 03**	1.193e + 03●	0.712	↑ *m*	**1.198e + 03**	1.194e + 03●	0.685	↑ *m*
n30Model4	**1.602e + 03**	1.595e + 03●	0.693	↑ *m*	**1.601e + 03**	1.595e + 03●	0.688	↑ *m*
n30Model5	**1.019e + 02**	1.010e + 03●	0.681	↑ *m*	**1.016e + 03**	1.007e + 03●	0.674	↑ *m*
n30Model6	**1.221e + 03**	1.216e + 03●	0.698	↑ *m*	**1.220e + 03**	1.216e + 03●	0.705	↑ *m*
n30Model7	**1.485e + 03**	1.473e + 03●	0.661	↑ *s*	**1.482e + 03**	1.473e + 03●	0.661	↑ *s*
n30Model8	**1.062e + 03**	1.055e + 03●	0.692	↑ *m*	**1.061e + 03**	1.055e + 03●	0.686	↑ *m*
n30Model9	**1.298e + 03**	1.285e + 03●	0.721	↑ *m*	**1.293e + 03**	1.285e + 03●	0.714	↑ *m*
n30Model10	**1.337e + 03**	1.325e + 03●	0.706	↑ *m*	**1.333e + 03**	1.321e + 03●	0.703	↑ *m*
n50Model1	**1.038e + 03**	1.031e + 03●	0.750	↑ *l*	**1.035e + 03**	1.031e + 03●	0.702	↑ *m*
n50Model2	**1.573e + 03**	1.558e + 03∗	0.641	↑ *s*	**1.567e + 03**	1.556e + 03∗	0.641	↑ *s*
n50Model3	**1.573e + 03**	1.556e + 03∗	0.641	↑ *s*	**1.568e + 03**	1.556e + 03∗	0.641	↑ *s*
n50Model4	**1.871e + 03**	1.840e + 03∗	0.623	↑ *s*	**1.857e + 03**	1.842e + 03∗	0.623	↑ *s*
n50Model5	**1.680e + 03**	1.667e + 03●	0.702	↑ *m*	**1.677e + 03**	1.668e + 03●	0.702	↑ *m*
n50Model6	**1.882e + 03**	1.868e + 03●	0.681	↑ *m*	**1.877e + 03**	1.867e + 03●	0.681	↑ *m*
n50Model7	**2.423e + 03**	2.405e + 03∗	0.628	↑ *s*	**2.419e + 03**	2.406e + 03∗	0.628	↑ *s*
n50Model8	**1.791e + 03**	1.777e + 03●	0.656	↑ *s*	**1.788e + 03**	1.778e + 03●	0.656	↑ *s*
n50Model9	**1.391e + 03**	1.377e + 03●	0.724	↑ *m*	**1.387e + 03**	1.377e + 03●	0.724	↑ *m*
n50Model10	**1.402e + 03**	1.388e + 03∗	0.630	↑ *s*	**1.397e + 03**	1.389e + 03∗	0.630	↑ *s*

the common characteristic of the preceding four models is that the number of constraints is large, leading to a long time to generate valid test cases during the search. Moreover, the number of 2-sets is also high, and thus the evaluation of the fitness function (i.e., 2-wise coverage) will be expensive. Both of the preceding could explain why the time cost is high on these FMs. On the remaining FMs, the runtime of MAP-Elites is 0.7 seconds at least and 94.1 seconds at most. The median is 9.6, meaning that half of the FMs can be handled within 9.6 seconds.

Moreover, Figure 7 shows the runtime comparison between MAP-Elites and MI-GA on all the FMs in Table 3. As can be seen, the ratio ranges from 0.75 to 1.26, indicating that the two algorithms

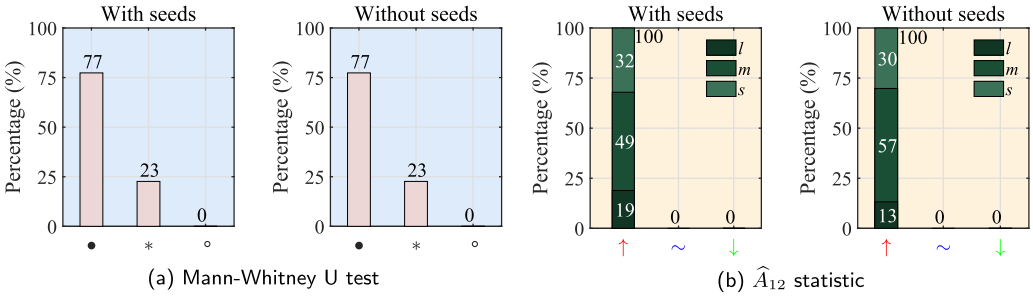


Fig. 4. Summary of the Mann-Whitney U test and $\widehat{A}_{12}$ statistic for the comparison between MAP-Elites and MI-GA regarding the QD-Score for 2-wise coverage reported in Table 3.

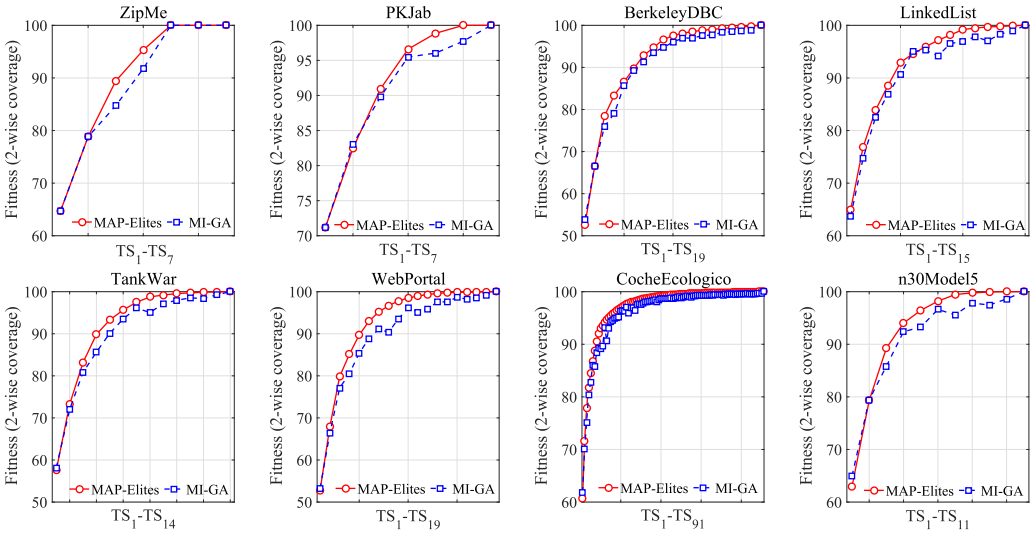


Fig. 5. The fitness value (2-wise coverage) of test suites returned by MAP-Elites and MI-GA in a particular run where the QD-Score of the fitness is closest to the mean over all 30 runs.

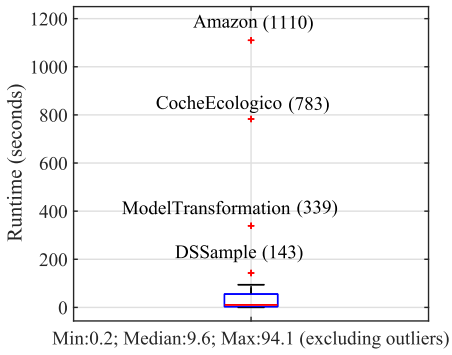


Fig. 6. Runtime (in seconds) of MAP-Elites.

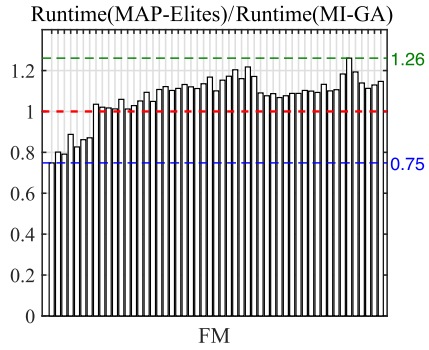


Fig. 7. Runtime comparisons between MAP-Elites and MI-GA. The FMs in the *x*-axis are arranged in the same order as in Table 3.

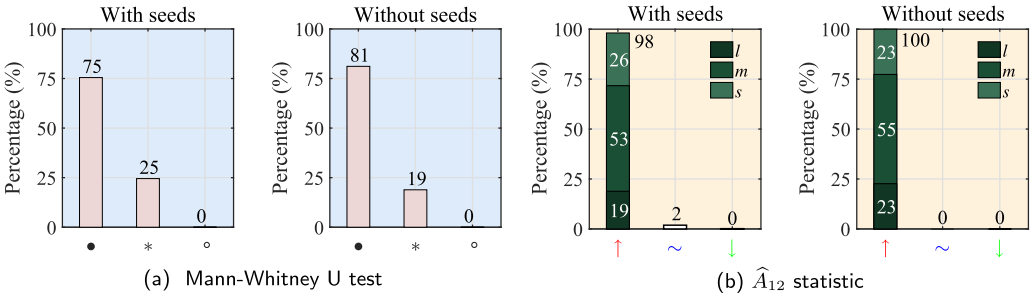


Fig. 8. Summary of the Mann-Whitney U test and $\widehat{A}_{12}$ statistic for the comparison between MAP-Elites and MI-GA regarding the QD-Score for test suite diversity.

have small differences regarding the computational efficiency. The underlying reason is that most parts of the two algorithms, except for the mutation, are implemented by the same code. For the mutation, MI-GA uses only test case substitution, whereas MAP-Elites uses two additional operators: test case addition and test case removal. According to Algorithm 3, however, each time, the scheduling of the three operators in MAP-Elites requires to generate a random number (line 17 in Algorithm 3), which results in additional computational burden. This may explain why MAP-Elites runs (slightly) longer than MI-GA.

6.1.2 Test Suite Diversity as the Fitness Function. Now we examine how both algorithms perform if the fitness function is changed to another one. To this end, we switch from 2-wise coverage to test suite diversity given by Equation (3). Experimental results (which are provided online) show that, no matter if seeds are used or not, MAP-Elites obtains a better QD-Score than MI-GA on all the considered FMs. According to Figure 8(a), MAP-Elites performs either significantly better than or statistically equivalently to MI-GA. Specifically, MAP-Elites significantly outperforms MI-GA on 75% and 81% of the FMs for `With seeds` and `Without seeds`, respectively. On the remaining FMs, differences between the two algorithms are found to be statistically insignificant. The $\widehat{A}_{12}$ statistic, as summarized in Figure 8(b), confirms the superiority of MAP-Elites when seeds are implanted, showing large, medium, and small differences in its favor on 19%, 53%, and 26% of the FMs, respectively. Furthermore, there are no FMs on which MI-GA is favored and only 2% FMs on which the two algorithms show negligible differences. Considering the scenario where seeds are not used, the $\widehat{A}_{12}$ results also confirm the superiority of MAP-Elites in this case, with large, medium, and small differences in its favor on 23%, 55%, and 23% of the FMs, respectively. Therefore, MAP-Elites

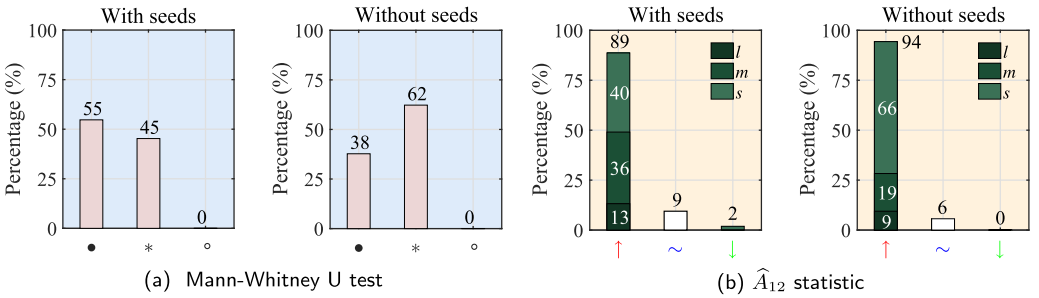


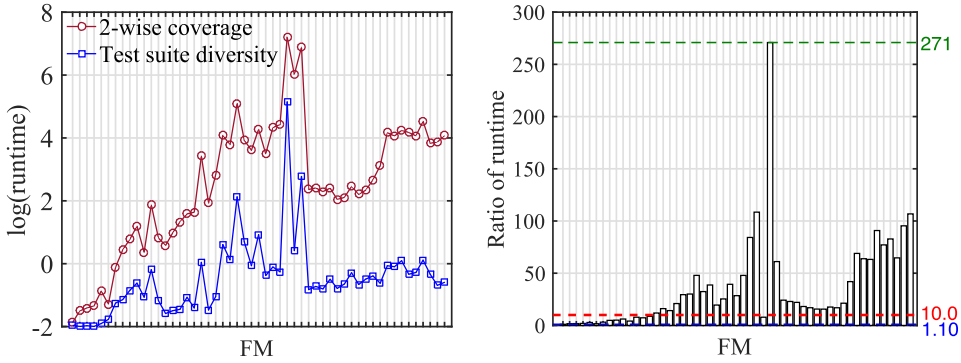
Fig. 9. Summary of the Mann-Whitney U test and $\widehat{A}_{12}$ statistic for the QD-Score comparison between MAP-Elites and MI-GA regarding 2-wise coverage of test suites that are generated by maximizing test suite diversity.

also shows significant and substantial improvements over MI-GA when test suite diversity is used as the fitness function. This implies that the superiority of MAP-Elites may not be affected by the adopted fitness functions.

As discussed previously, the hypothesis of using test suite diversity as fitness is that diverse test suites could potentially lead to high t-wise coverage. In the following, we will verify this hypothesis by reporting 2-wise coverage for test suites that are generated by maximizing test suite diversity. Experimental results (which are provided online) show that even though 2-wise coverage is not directly optimized, MAP-Elites is able to obtain higher 2-wise coverage than MI-GA on the majority of the FMs. We are also aware that there are some exceptions (e.g., PKJab) on which MI-GA obtains a better QD-Score than MAP-Elites. However, differences between the two algorithms in all the exceptional cases are not statistically significant. As before, Figure 9 summarizes Mann-Whitney U tests and $\widehat{A}_{12}$ statistic results. According to Figure 9(a), MAP-Elites performs significantly better than or equivalently to MI-GA on all the FMs. Figure 9(b) further confirms the superiority of MAP-Elites, showing that MAP-Elites is favored on 89% of the FMs for `With seeds`, and 94% for `Without seeds`. According to the preceding experimental results, we can find that comparisons made based on 2-wise coverage are generally consistent with those made based on test suite diversity. In fact, this phenomenon is not surprising and can be well explained. As observed and discussed in our previous work [96], there exist significant positive correlations between test suite diversity and t-wise coverage, and thus optimizing test suite diversity will indirectly improve t-wise coverage. However, test suite diversity is much computationally cheaper than t-wise coverage, which is clearly shown in Figure 10. As can be seen, the ratio of runtime for 2-wise coverage to that for test suite diversity is at least 1.1 and at most 271. On the majority of the FMs, the ratio exceeds 10. Therefore, the use of test suite diversity is advisable for large FMs (as will be done in the experiments for RQ4).

6.1.3 Explanation and Summary. In the preceding experiments, we show that MAP-Elites overwhelmingly outperforms MI-GA in terms of the overall performance (measured by the QD-Score), regardless of which fitness functions are adopted, and whether or not seeds are implanted. Notice that all preceding experimental results are obtained by setting ub to the test suite size of SamplingCA, lb to $\max\{2, ub - 99\}$, and the number of FEs to $500 \times (ub - lb + 1)$, following the specifications in Section 5.3. Due to limited space, experimental results with $lb = ub - 5$[16] and

[16]This means that the archive size (i.e., $ub - lb + 1$) is 6. We notice that 7 is the minimum size of the test suites found by SamplingCA on all the FMs. Therefore, starting from the size 2 and ending with the size 7, there are exactly six available test suite sizes.



(a) Logarithmic runtime (measured in seconds) (b) Runtime(Coverage)/Runtime(Diversity)

Fig. 10. Runtime comparisons between 2-wise coverage and test suite diversity.

$FEs = 2{,}000 \times 6$ are not provided here but are found in the online supplement. Interestingly, similar conclusions can be drawn from those results. Therefore, the superiority of MAP-Elites over MI-GA is also unlikely affected by the settings of the archive size (determined by ub and lb) and the termination conditions (specified by the number of FEs).

Going one step further, we will explore the underlying reasons for the high performance of MAP-Elites. Recall that MAP-Elites enables information sharing between QD subproblems due to the simultaneous use of three mutation operators: test case removal, test case addition, and test case substitution. In contrast, since the mechanism in MI-GA is to search for the best test suites independently, it uses only test case substitution, thus without information sharing. We argue that information sharing realized by test case removal and test case addition is the key to the success of MAP-Elites. To verify this, Figure 11 shows the number of successful updates for the three mutation operators in a typical run of MAP-Elites. In this experiment, 2-wise coverage is used as the fitness function, and a success update means that a better test suite with higher 2-wise coverage is found. According to Figure 11, for each FM and each test suite in the archive, test case removal and test case addition generally lead to (much) more successful updates than test case substitution. Notably, for the 4th test suite on `BerkeleyDBFootprint`, the 2nd test suite on `HiPAcc`, and the 3rd and 7th test suites on `n50Model1`, substitution does not contribute to any fitness improvement. It should be mentioned that similar observations are found on other FMs, not only those chosen in Figure 11. The preceding results clearly demonstrate the importance of test case removal and test case addition. In fact, with the two mutation operators, good solution patterns can be shared by neighboring subproblems. Therefore, they are optimized in a cooperative way. However, MI-GA disables information sharing, and thus good solutions found for one subproblem cannot be reused as stepping stones by others. As a result, it will slow down the optimization process on the whole, and may also cause performance fluctuations. Indeed, as can be seen from Figure 5, the 2-wise coverage within a single archive returned by MI-GA fluctuates on some FMs (e.g., `LinkedList` and `n30Model5`).

As a summary of Section 6.1, we have the following conclusions. First, MAP-Elites shows significant and substantial improvements over MI-GA on most of the FMs under study. Second, the superiority of MAP-Elites over MI-GA is insensitive to which fitness functions are used (2-wise coverage or test suite diversity), and whether the initial population is seeded or not. Third, the great success of MAP-Elites is attributed to the information sharing brought by the simultaneous use of three mutation operators. In particular, we find that test case removal and test case addition

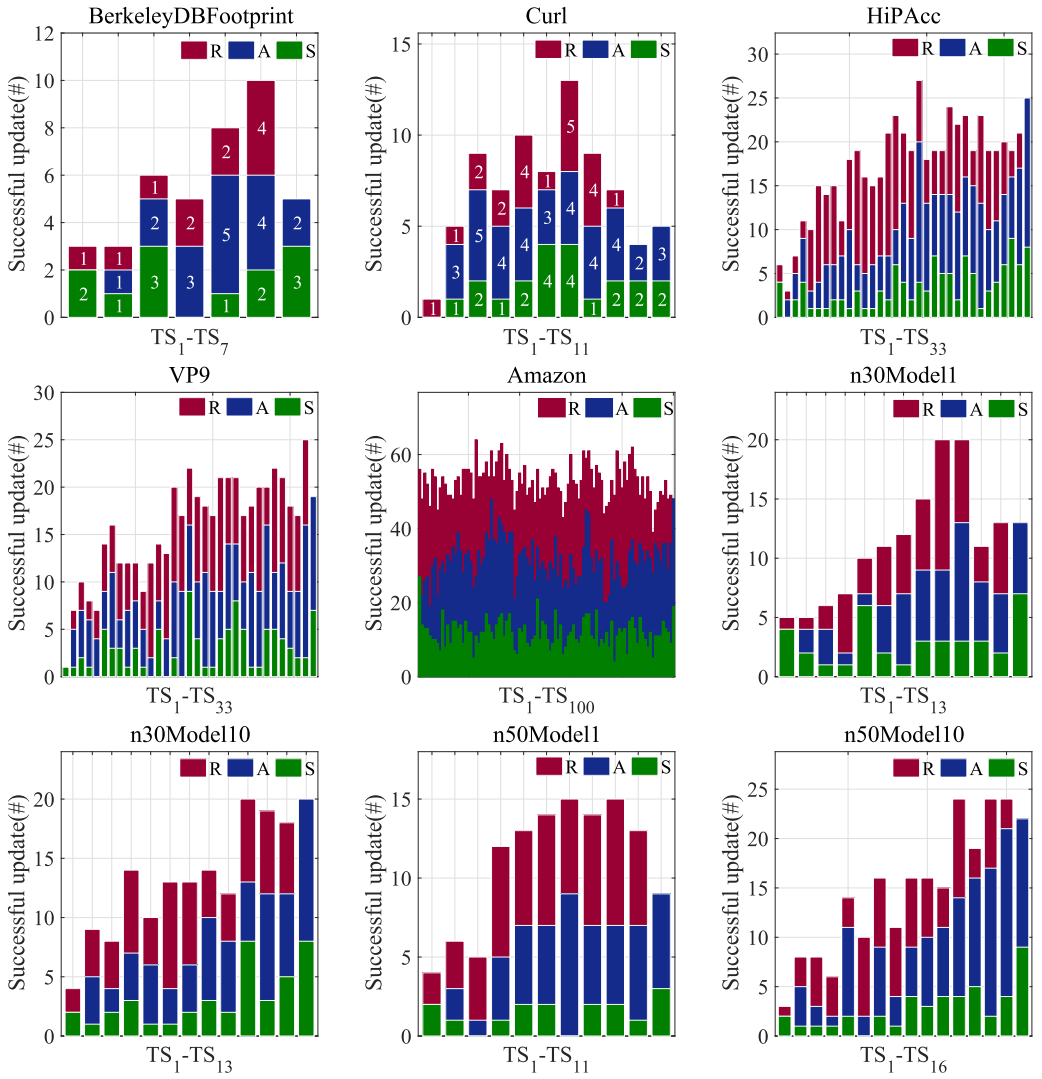


Fig. 11. Number of successful updates for test case removal (R), test case addition (A), and test case substitution (S) on nine representative FMs in a typical run of MAP-Elites.

contribute more to fitness improvement than test case substitution. Test case removal and test case addition allow the exchange of good solution patterns that can be reused as stepping stones for neighboring subproblems.

6.2 RQ2: Comparison with NSGA-II

Table 4 gives the QD-Score for the pairwise coverage obtained by MAP-Elites and NSGA-II. Figure 12(a) and (b) summarize the Mann-Whitney U results and $\widehat{A}_{12}$ statistics, respectively. According to Figure 12(a), MAP-Elites performs significantly better than or equivalently to NSGA-II on all FMs. Regarding $\widehat{A}_{12}$ results, as shown in Figure 12(b), large, medium, and small differences in favor of MAP-Elites are 94%, 4%, and 0%, respectively. There are no cases in which

Table 4. QD-Score for Pairwise Coverage Obtained by MAP-Elites and NSGA-II

	MAP-Elites	NSGA-II	$\widehat{A}_{12}$	
ZipMe	**6.318e + 02**	**6.318e + 02**∗	0.502	$\sim n$
BerkeleyDBFootprint	**6.070e + 02**	5.164e + 02•	0.759	$\uparrow l$
Apache	**6.179e + 02**	5.241e + 02•	0.756	$\uparrow l$
argo-uml-spl	**6.270e + 02**	5.276e + 02•	0.843	$\uparrow l$
LLVM	**7.095e + 02**	6.095e + 02•	0.704	$\uparrow m$
PKJab	**6.444e + 02**	5.469e + 02•	0.799	$\uparrow l$
Curl	**9.824e + 02**	7.818e + 02•	0.864	$\uparrow l$
Wget	**9.757e + 02**	7.760e + 02•	0.877	$\uparrow l$
x264	**1.351e + 03**	1.150e + 03•	0.917	$\uparrow l$
BerkeleyDBC	**1.724e + 03**	1.427e + 03•	1.000	$\uparrow l$
gpl	**1.114e + 03**	8.234e + 02•	0.859	$\uparrow l$
BerkeleyDBMemory	**2.653e + 03**	2.159e + 03•	1.000	$\uparrow l$
fame_dbms_fm	**1.200e + 03**	9.544e + 02•	0.861	$\uparrow l$
DesktopSearcher	**8.228e + 02**	7.228e + 02•	0.824	$\uparrow l$
CounterStrikeSimpleFM	**9.076e + 02**	7.104e + 02•	0.904	$\uparrow l$
BerkeleyDBPerformance	**9.148e + 02**	8.127e + 02•	0.729	$\uparrow m$
LinkedList	**1.389e + 03**	1.092e + 03•	0.987	$\uparrow l$
SensorNetwork	**9.879e + 02**	7.891e + 02•	0.902	$\uparrow l$
HiPAcc	**3.057e + 03**	2.568e + 03•	1.000	$\uparrow l$
SPLSSimuelESPnP	**1.007e + 03**	8.077e + 02•	0.943	$\uparrow l$
TankWar	**1.281e + 03**	1.172e + 03•	0.923	$\uparrow l$
JavaGC	**4.103e + 03**	3.559e + 03•	1.000	$\uparrow l$
Polly	**3.072e + 03**	2.577e + 03•	0.986	$\uparrow l$
DSSample	**8.830e + 03**	7.104e + 03•	1.000	$\uparrow l$
VP9	**3.022e + 03**	2.574e + 03•	0.987	$\uparrow l$
WebPortal	**1.651e + 03**	1.449e + 03•	0.917	$\uparrow l$
JHipster	**3.478e + 03**	2.982e + 03•	1.000	$\uparrow l$
Drupal	**1.275e + 03**	1.065e + 03•	0.994	$\uparrow l$
SmartHomev2.2	**1.462e + 03**	1.257e + 03•	0.974	$\uparrow l$
VideoPlayer	**1.286e + 03**	1.041e + 03•	0.978	$\uparrow l$
Amazon	**9.813e + 03**	6.666e + 03•	1.000	$\uparrow l$
ModelTransformation	**2.659e + 03**	2.307e + 03•	0.990	$\uparrow l$
CocheEcologico	**8.741e + 03**	7.192e + 03•	1.000	$\uparrow l$
n30Model1	**1.105e + 03**	9.533e + 02•	0.942	$\uparrow l$
n30Model2	**1.306e + 03**	1.107e + 03•	0.916	$\uparrow l$
n30Model3	**1.198e + 03**	1.000e + 03•	0.908	$\uparrow l$
n30Model4	**1.601e + 03**	1.352e + 03•	0.947	$\uparrow l$
n30Model5	**1.016e + 03**	9.133e + 02•	0.891	$\uparrow l$
n30Model6	**1.220e + 03**	1.020e + 03•	0.912	$\uparrow l$
n30Model7	**1.482e + 03**	1.234e + 03•	0.930	$\uparrow l$
n30Model8	**1.061e + 03**	9.122e + 02•	0.874	$\uparrow l$
n30Model9	**1.293e + 03**	1.093e + 03•	0.962	$\uparrow l$
n30Model10	**1.333e + 03**	1.084e + 03•	0.963	$\uparrow l$
n50Model1	**1.035e + 03**	9.327e + 02•	0.964	$\uparrow l$
n50Model2	**1.567e + 03**	1.273e + 03•	0.908	$\uparrow l$
n50Model3	**1.568e + 03**	1.365e + 03•	0.898	$\uparrow l$
n50Model4	**1.857e + 03**	1.655e + 03•	0.908	$\uparrow l$
n50Model5	**1.677e + 03**	1.477e + 03•	0.959	$\uparrow l$
n50Model6	**1.877e + 03**	1.676e + 03•	0.956	$\uparrow l$
n50Model7	**2.419e + 03**	2.170e + 03•	0.914	$\uparrow l$
n50Model8	**1.788e + 03**	1.586e + 03•	0.970	$\uparrow l$
n50Model9	**1.387e + 03**	1.184e + 03•	0.966	$\uparrow l$
n50Model10	**1.397e + 03**	1.244e + 03•	0.877	$\uparrow l$

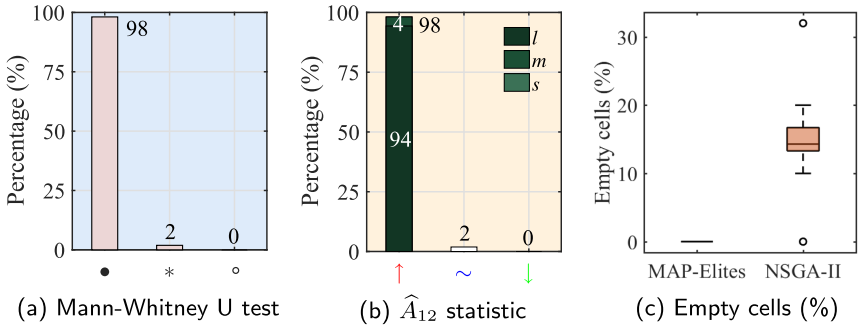


Fig. 12. Comparisons between MAP-Elites and NSGA-II.

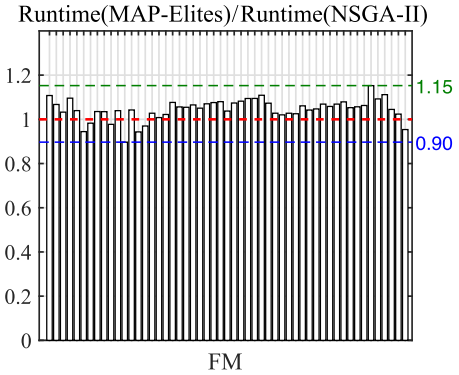


Fig. 13. Runtime comparisons between MAP-Elites and NSGA-II.

NSGA-II is favored. The preceding results clearly highlight the superiority of MAP-Elites over NSGA-II. This conclusion is further confirmed by the online experimental results with different settings for the test suite size and termination condition. Concerning the computational efficiency, according to Figure 13, the ratio of runtime ranges from 0.9 to 1.15, suggesting that MAP-Elites is in general as efficient as NSGA-II.

Now we explain why NSGA-II is not as effective as MAP-Elites in illuminating the behavior space. As mentioned previously, solutions of NSGA-II may not be able to fill all the slots in the archive. To show this, we plot in Figure 12(c) the average percentage of empty cells in the form of boxplots for all FMs. For MAP-Elites, it is not surprising that there are no empty cells since it is designed to fill all the slots. For NSGA-II, there are indeed empty cells on some FMs. Concerning the percentage, the minimum, median, and maximum values are 0%, 14.3%, and 32%, respectively. We argue that global competitions (or unconstrained competitions) in NSGA-II are the main reason for empty cells. In fact, the Pareto dominance used in NSGA-II compares solutions considering all the objectives regardless of which cell the solution is located in. Then, it is possible that a high-performing solution Pareto dominates its neighboring solutions. Thus, these neighboring solutions could be potentially removed in the environmental selection, leaving the corresponding cells unoccupied. In contrast, competitions in MAP-Elites are restricted in the same cells. This is to say, a solution is not allowed to replace its neighboring solutions even though it has a better fitness value. Also due to unconstrained competitions, multiple (reduplicated) solutions within the same cell can possibly survive during the environmental selection of NSGA-II as they are non-dominated with each other. As a natural consequence, there must be some cells for which no

solution is selected. Different from NSGA-II, MAP-Elites always maintains at most one solution for each cell. This mechanism guarantees that, in the context of this article, each cell is infilled in MAP-Elites.

We should also mention that a prerequisite of using NSGA-II (or a general MOEA) is that the optimization objectives are conflicting. Although this is true in our context (i.e., minimizing test suite size is conflicting with maximizing pairwise coverage [61]), it is not always the case in other circumstances. For example, if we use a new behavior, such as the mutation score [41], which may be positively correlated with the 2-wise coverage [34], then NSGA-II may not be a good choice as it tends to generate a single solution instead of a set of solutions. In contrast, MAP-Elites always generates a set of solutions no matter whether or not the objective is conflicting with the behavior(s).

To summarize, both Mann-Whitney U and $\widehat{A}_{12}$ tests suggest that MAP-Elites significantly outperforms NSGA-II regarding the overall performance of the returned test suites. The reason for the relatively poor performance of NSGA-II is that, due to unconstrained competitions, NSGA-II fails to generate solutions for some cells in the archive, even though these cells can be possibly filled with some solution. Therefore, we can conclude that MAP-Elites (or broadly QD algorithms) is more effective than NSGA-II (or broadly MOEAs) when generating a large set of diverse and high-performing test suites in SPL testing.

6.3 RQ3: Comparison with State-of-the-Art Testing Tools

Table 5 gives experimental results for MAP-Elites and three state-of-the-art t-wise testing tools: i.e., YASA, IncLing, and SamplingCA. Figure 14 summarizes $\widehat{A}_{12}$ statistics regarding pairwise comparisons between MAP-Elites and each of the preceding tools. According to Figure 14(a), differences between MAP-Elites and YASA are large on all FMs. Specifically, large effect sizes in favor of MAP-Elites and YASA are 96% and 4%, respectively. Compared with IncLing, as can be seen from Figure 14(b), MAP-Elites performs better on 58% of FMs and worse on 40% of FMs. Finally, Figure 14(c) shows that MAP-Elites performs worse than SamplingCA on 89% of FMs, with large, medium, and small differences in favor of SamplingCA on 28%, 42%, and 19% of FMs, respectively. The preceding comparisons suggest that, considering the overall performance, MAP-Elites performs (significantly) better than YASA and IncLing but worse than SamplingCA. We give in Figure 15(a) the runtime (seconds) for the three testing tools, showing that SamplingCA is faster than IncLing, then YASA. It is found that the faster the testing tool, the worse the performance of MAP-Elites in comparison with that tool. Because MAP-Elites is a search-based approach and it uses the same time budget as the testing tools, it tends to do better if more time is given. This in turn explains why MAP-Elites is inferior to SamplingCA. According to Luo et al. [67], SamplingCA is highly efficient, running one to two orders of magnitude faster than state-of-the-art tools. This is confirmed by our experiments results, which show that, in the best cases, SamplingCA is 75 and 35 times faster than YASA and IncLing, respectively. It is highly possible that the search in MAP-Elites is insufficient due to limited search budget. Moreover, SamplingCA incrementally appends the best test case among k (a hyperparameter, e.g., 1,000) candidates to the test suite to be returned [67]. This makes SamplingCA effective in generating high-performing derived test suites (recall that derived test suites are generated by removing the last test case one by one).

However, this does not mean the uselessness of MAP-Elites. Instead, as a search-based approach, it can serve as a potential algorithm to reduce the size of covering arrays returned by these t-wise testing tools. To evaluate the effectiveness of MAP-Elites as such an algorithm, we focus on how often the initial covering array (called *seed*) is reduced. Note that a run is deemed successful if MAP-Elites finds a smaller covering array than YASA, IncLing, or SamplingCA. Table 6 tabulates the percentage of successful runs for MAP-Elites (as well as GrES). As can be seen, MAP-Elites is

Table 5. QD-Score for Pairwise Coverage Obtained by MAP-Elites, YASA, IncLing, and SamplingCA

	MAP-Elites	YASA	$\widehat{A}_{12}$	MAP-Elites	IncLing	$\widehat{A}_{12}$	MAP-Elites	SamplingCA	$\widehat{A}_{12}$
ZipMe	**5.306e + 02**	4.682e + 02●	↑ *l*	**7.306e + 02**	7.071e + 02●	↑ *l*	**6.218e + 02**	6.106e + 02●	↑ *l*
BerkeleyDBFootprint	**5.035e + 02**	4.867e + 02●	↑ *l*	**6.063e + 02**	5.844e + 02●	↑ *l*	**5.926e + 02**	5.844e + 02●	↑ *l*
Apache	**6.145e + 02**	5.862e + 02●	↑ *l*	**6.172e + 02**	5.979e + 02●	↑ *l*	**6.093e + 02**	5.979e + 02●	↑ *l*
argo-uml-spl	**5.248e + 02**	4.926e + 02●	↑ *l*	**6.248e + 02**	6.092e + 02●	↑ *l*	**6.172e + 02**	6.092e + 02●	↑ *l*
LLVM	**7.088e + 02**	6.516e + 02●	↑ *l*	**7.102e + 02**	6.869e + 02●	↑ *l*	**6.984e + 02**	6.873e + 02●	↑ *m*
PKJab	**5.395e + 02**	5.136e + 02●	↑ *l*	**5.395e + 02**	5.305e + 02●	↑ *l*	**6.325e + 02**	6.316e + 02∗	↑ *s*
Curl	**9.764e + 02**	9.482e + 02●	↑ *l*	**1.476e + 03**	1.454e + 03●	↑ *l*	9.632e + 02	**9.691e + 02**○	↓ *l*
Wget	**9.677e + 02**	9.229e + 02●	↑ *l*	**1.269e + 03**	1.254e + 03●	↑ *l*	9.579e + 02	**9.644e + 02**○	↓ *m*
x264	**1.444e + 03**	1.342e + 03●	↑ *l*	**1.845e + 03**	1.821e + 03●	↑ *l*	1.327e + 03	**1.347e + 03**○	↓ *l*
BerkeleyDBC	**1.813e + 03**	1.753e + 03●	↑ *l*	**2.013e + 03**	1.994e + 03●	↑ *l*	1.686e + 03	**1.723e + 03**○	↓ *l*
gpl	**1.006e + 03**	9.299e + 02●	↑ *l*	**1.108e + 03**	1.101e + 03●	↑ *l*	1.079e + 03	**1.097e + 03**○	↓ *m*
BerkeleyDBMemory	**2.624e + 03**	2.550e + 03●	↑ *l*	**2.828e + 03**	2.805e + 03●	↑ *l*	2.584e + 03	**2.654e + 03**○	↓ *l*
fame_dbms_fm	**9.947e + 02**	9.180e + 02●	↑ *l*	**1.593e + 03**	1.581e + 03●	↑ *l*	1.180e + 03	**1.190e + 03**○	↓ *m*
DesktopSearcher	**8.175e + 02**	7.408e + 02●	↑ *l*	**9.177e + 02**	9.110e + 02●	↑ *l*	8.073e + 02	**8.182e + 02**○	↓ *l*
CounterStrikeSFM	**8.008e + 02**	7.938e + 02●	↑ *l*	**8.999e + 02**	8.904e + 02●	↑ *l*	8.877e + 02	**9.048e + 02**○	↓ *l*
BerkeleyDBP	**8.092e + 02**	7.567e + 02●	↑ *l*	**1.010e + 03**	9.957e + 02●	↑ *l*	9.022e + 02	**9.088e + 02**○	↓ *m*
LinkedList	**1.377e + 03**	1.301e + 03●	↑ *l*	1.875e + 03	**1.878e + 03**○	↓ *l*	1.354e + 03	**1.388e + 03**○	↓ *l*
SensorNetwork	**1.073e + 03**	1.011e + 03●	↑ *l*	**1.375e + 03**	1.359e + 03●	↑ *l*	9.581e + 02	**9.792e + 02**○	↓ *m*
HiPAcc	**3.109e + 03**	3.067e + 03●	↑ *l*	**3.715e + 03**	3.711e + 03●	↑ *m*	2.951e + 03	**3.043e + 03**○	↓ *m*
SPLSSimuelESPnP	**8.049e + 02**	7.525e + 02●	↑ *l*	**1.604e + 03**	1.603e + 03∗	↑ *s*	9.878e + 02	**1.003e + 03**○	↓ *l*
TankWar	**1.367e + 03**	1.302e + 03●	↑ *l*	**1.568e + 03**	1.568e + 03∗	~ *n*	1.260e + 03	**1.283e + 03**○	↓ *m*
JavaGC	**4.144e + 03**	4.090e + 03●	↑ *l*	5.050e + 03	**5.074e + 03**○	↓ *l*	4.014e + 03	**4.115e + 03**∗	↓ *s*
Polly	**3.334e + 03**	3.256e + 03●	↑ *l*	3.539e + 03	**3.550e + 03**○	↓ *l*	3.009e + 03	**3.076e + 03**∗	↓ *s*
DSSample	8.632e + 03	**8.723e + 03**○	↓ *l*	9.457e + 03	**9.613e + 03**○	↓ *l*	8.508e + 03	**8.869e + 03**○	↓ *l*
VP9	**3.238e + 03**	3.191e + 03●	↑ *l*	3.544e + 03	**3.545e + 03**∗	↓ *s*	2.973e + 03	**3.028e + 03**∗	↓ *s*
WebPortal	**1.821e + 03**	1.729e + 03●	↑ *l*	2.421e + 03	**2.430e + 03**○	↓ *l*	1.615e + 03	**1.660e + 03**○	↓ *m*
JHipster	**3.332e + 03**	3.161e + 03●	↑ *l*	4.627e + 03	**4.653e + 03**○	↓ *l*	3.312e + 03	**3.397e + 03**○	↓ *l*
Drupal	**1.162e + 03**	1.116e + 03●	↑ *l*	1.266e + 03	**1.268e + 03**○	↓ *l*	1.245e + 03	**1.275e + 03**○	↓ *l*
SmartHomev2.2	**1.544e + 03**	1.460e + 03●	↑ *l*	1.745e + 03	**1.755e + 03**○	↓ *l*	1.433e + 03	**1.466e + 03**○	↓ *m*
VideoPlayer	**1.081e + 03**	9.969e + 02●	↑ *l*	1.282e + 03	**1.289e + 03**○	↓ *l*	1.269e + 03	**1.289e + 03**○	↓ *m*
Amazon	9.674e + 03	**9.842e + 03**○	↓ *l*	9.814e + 03	**9.946e + 03**○	↓ *l*	9.674e + 03	**9.927e + 03**○	↓ *l*
ModelTransformation	**2.983e + 03**	2.844e + 03●	↑ *l*	3.190e + 03	**3.214e + 03**○	↓ *l*	2.623e + 03	**2.674e + 03**○	↓ *m*
CocheEcologico	**8.507e + 03**	8.479e + 03●	↑ *l*	9.132e + 03	**9.220e + 03**○	↓ *l*	8.629e + 03	**8.760e + 03**○	↓ *l*
n30Model1	**1.094e + 03**	9.976e + 02●	↑ *l*	**1.296e + 03**	1.276e + 03●	↑ *l*	1.083e + 03	**1.101e + 03**○	↓ *l*
n30Model2	**1.398e + 03**	1.315e + 03●	↑ *l*	**1.701e + 03**	1.684e + 03●	↑ *l*	1.276e + 03	**1.301e + 03**○	↓ *m*
n30Model3	**1.387e + 03**	1.324e + 03●	↑ *l*	**1.589e + 03**	1.577e + 03●	↑ *l*	1.176e + 03	**1.199e + 03**○	↓ *m*
n30Model4	**1.482e + 03**	1.413e + 03●	↑ *l*	**2.180e + 03**	2.171e + 03●	↑ *l*	1.553e + 03	**1.590e + 03**○	↓ *m*
n30Model5	**1.004e + 03**	9.513e + 02●	↑ *l*	**1.305e + 03**	1.284e + 03●	↑ *l*	9.844e + 02	**1.014e + 03**○	↓ *m*
n30Model6	**1.115e + 03**	1.085e + 03●	↑ *l*	**1.514e + 03**	1.508e + 03●	↑ *l*	1.196e + 03	**1.219e + 03**○	↓ *m*
n30Model7	**1.558e + 03**	1.488e + 03●	↑ *l*	2.060e + 03	**2.067e + 03**○	↓ *l*	1.439e + 03	**1.484e + 03**○	↓ *s*
n30Model8	**1.003e + 03**	9.014e + 02●	↑ *l*	**1.304e + 03**	1.287e + 03●	↑ *l*	1.022e + 03	**1.056e + 03**○	↓ *m*
n30Model9	**1.376e + 03**	1.337e + 03●	↑ *l*	**1.678e + 03**	1.674e + 03●	↑ *m*	1.241e + 03	**1.293e + 03**○	↓ *m*
n30Model10	**1.367e + 03**	1.275e + 03●	↑ *l*	**1.771e + 03**	1.758e + 03●	↑ *l*	1.286e + 03	**1.334e + 03**○	↓ *m*
n50Model1	**9.301e + 02**	8.752e + 02●	↑ *l*	**9.311e + 02**	9.289e + 02●	↑ *l*	1.022e + 03	**1.035e + 03**○	↓ *l*
n50Model2	**1.455e + 03**	1.344e + 03●	↑ *l*	1.953e + 03	**1.962e + 03**○	↓ *l*	1.527e + 03	**1.571e + 03**∗	↓ *s*
n50Model3	**1.454e + 03**	1.344e + 03●	↑ *l*	1.954e + 03	**1.962e + 03**○	↓ *l*	1.534e + 03	**1.571e + 03**∗	↓ *s*
n50Model4	**1.830e + 03**	1.744e + 03●	↑ *l*	2.531e + 03	**2.543e + 03**○	↓ *l*	1.803e + 03	**1.867e + 03**∗	↓ *s*
n50Model5	**1.760e + 03**	1.575e + 03●	↑ *l*	2.557e + 03	**2.563e + 03**○	↓ *l*	1.637e + 03	**1.679e + 03**○	↓ *m*
n50Model6	**2.057e + 03**	1.959e + 03●	↑ *l*	2.260e + 03	**2.264e + 03**∗	↓ *s*	1.840e + 03	**1.884e + 03**○	↓ *m*
n50Model7	**2.435e + 03**	2.255e + 03●	↑ *l*	**2.642e + 03**	2.613e + 03●	↑ *l*	2.355e + 03	**2.421e + 03**∗	↓ *s*
n50Model8	**1.666e + 03**	1.632e + 03●	↑ *l*	2.263e + 03	**2.276e + 03**○	↓ *l*	1.746e + 03	**1.792e + 03**○	↓ *s*
n50Model9	**1.473e + 03**	1.417e + 03●	↑ *l*	1.676e + 03	**1.679e + 03**○	↓ *m*	1.353e + 03	**1.392e + 03**○	↓ *m*
n50Model10	**1.482e + 03**	1.335e + 03●	↑ *l*	**1.583e + 03**	1.578e + 03●	↑ *l*	1.372e + 03	**1.399e + 03**∗	↓ *s*

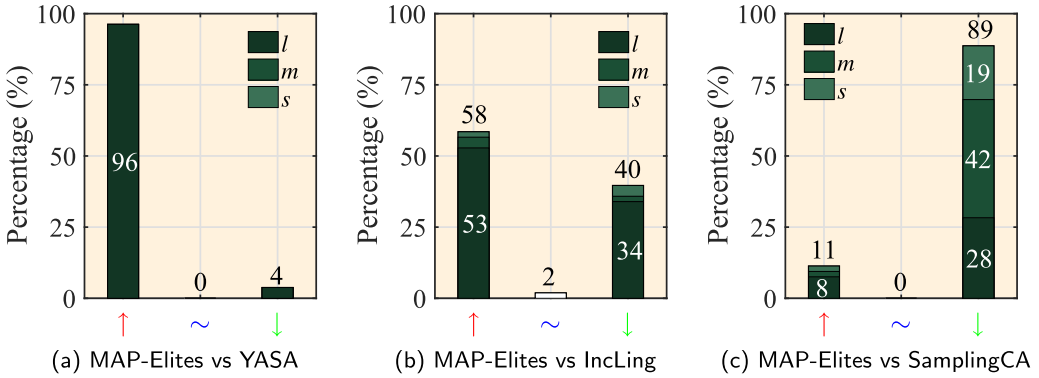


Fig. 14. $\widehat{A}_{12}$ comparisons between MAP-Elites and three state-of-the-art t-wise testing tools.

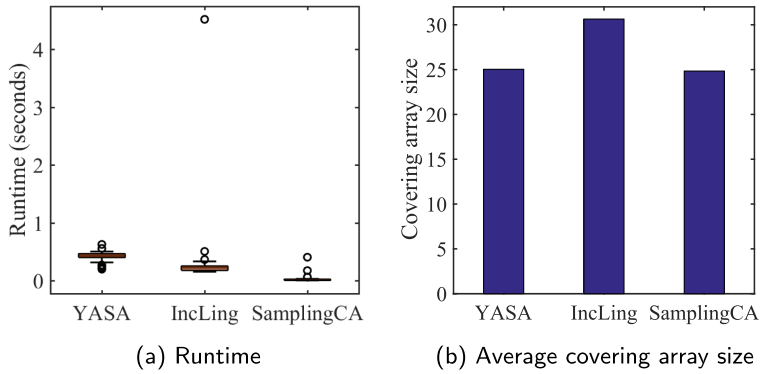


Fig. 15. Runtime and covering array size for the three state-of-the-art t-wise testing tools.

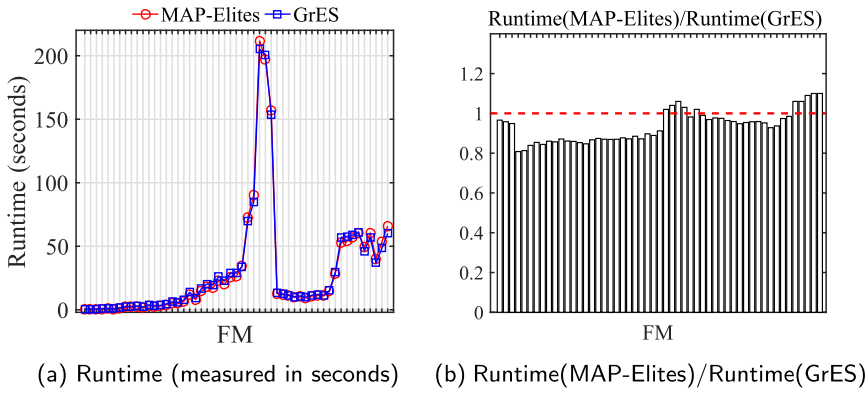
able to reduce sizes of the seeds generated by the three testing tools on most of the FMs. Notably, the percentage of successful runs can be as large as 100%. It is observed that covering arrays of IncLing are easier to be reduced than those of YASA and SamplingCA. Indeed, when improving the seeds of IncLing, the percentage of successful runs for MAP-Elites is 100% on a number of FMs. The underlying reason could be that compared with the covering arrays returned by YASA and SamplingCA, those found by IncLing are relatively large, which is shown in Figure 15(b). Intuitively, larger covering arrays are more likely to be improved than those smaller ones. In summary, the preceding experimental results suggest that covering arrays returned by YASA, IncLing, and SamplingCA are often not minimal and can be further reduced by using MAP-Elites.

Regarding comparisons between MAP-Elites and GrES, as can be seen from Table 6, MAP-Elites is much more effective than GrES, regardless of which seeding tool is used. Considering the seeding tools YASA, IncLing. and SamplingCA, the average percentage of successful runs for MAP-Elites is 43.21%, 88.18%, and 40.94%, respectively. In contrast, the corresponding value for GrES is only 7.36%, 68.49%, and 15.16%. In addition, MAP-Elites is able to reach 100% of successful runs on at least one FM for all three seeding tools, but this is not the case for GrES. The following would be one possible explanation for the ineffectiveness of GrES in reducing the covering array size. Motivated by the fact that pairwise coverage can be seen as the main aim in test suite generation, GrES emphasizes more on pairwise coverage than on other objective(s) [45]. Therefore, the goal of GrES is to generate test suites achieving full pairwise coverage and ideally also doing well for other objective(s). That is to say, in our context, test suite size is used as a secondary objective awaked only when pairwise coverage fails to distinguish between test suites. Pursuing high and even full pairwise coverage, however, tends to increase or retain, rather than reduce, the size of the current test suite. Indeed, adding a test case is more likely to improve pairwise coverage than removing or substituting a test case. Therefore, GrES is likely to be ineffective in reducing the size of covering arrays yielded by existing testing tools. Note that, as in MAP-Elites, GrES adopts the same three operations (i.e., test case removal, test case substitution, and test case addition) to mutate a test suite. As a result, MAP-Elites and GrES have similar computational efficiency, which is shown in Figure 16. For both algorithms, as can be seen, the runtime depends on the FMs to be handled. They finish the search within 1 second in the best case and around 200 seconds in the worst case.

Experimental results in this section emphasize the following. First, in the context of generating multiple high-performing test suites at a time, MAP-Elites performs better than YASA and IncLing but worse than SamplingCA. Second, as a search-based algorithm, MAP-Elites, which significantly outperforms GrES, is effective in reducing the size of covering arrays generated by YASA, IncLing,

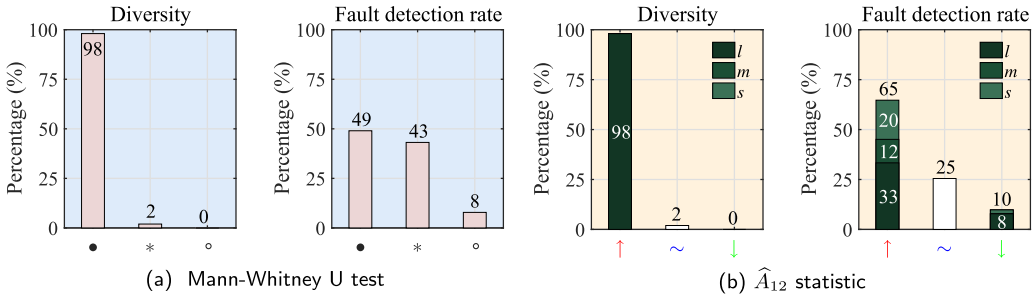
Table 6. Percentage of Successful Runs for MAP-Elites and GrES When Seeds Are Generated by Different t-Wise Testing Tools

	Seeds generated by YASA		Seeds generated by IncLing		Seeds generated by SamplingCA	
	MAP-Elites	GrES	MAP-Elites	GrES	MAP-Elites	GrES
ZipMe	**100.00%**	30.00%	**100.00%**	90.00%	**100.00%**	60.00%
BerkeleyDBFootprint	**16.67%**	3.33%	**70.00%**	16.67%	**76.67%**	33.33%
Apache	**86.67%**	20.00%	**83.33%**	36.67%	**80.00%**	13.33%
argo-uml-spl	**20.00%**	3.33%	**80.00%**	16.67%	**83.33%**	26.67%
LLVM	**96.67%**	30.00%	**96.67%**	16.67%	**80.00%**	30.00%
PKJab	**36.67%**	3.33%	**20.00%**	0.00%	**86.67%**	40.00%
Curl	**13.33%**	3.33%	**100.00%**	**100.00%**	**40.00%**	6.67%
Wget	**3.33%**	0.00%	**100.00%**	53.33%	**30.00%**	3.33%
x264	**33.33%**	3.33%	**100.00%**	96.67%	**23.33%**	13.33%
BerkeleyDBC	**100.00%**	30.00%	**100.00%**	90.00%	10.00%	**13.33%**
gpl	0.00%	0.00%	**100.00%**	36.67%	**46.67%**	20.00%
BerkeleyDBMemory	0.00%	0.00%	**100.00%**	86.67%	13.33%	**20.00%**
fame_dbms_fm	**26.67%**	0.00%	**100.00%**	**100.00%**	**70.00%**	13.33%
DesktopSearcher	**76.67%**	10.00%	**100.00%**	23.33%	**50.00%**	10.00%
CounterStrikeSimpleFM	0.00%	0.00%	**16.67%**	3.33%	**26.67%**	3.33%
BerkeleyDBPerformance	**26.67%**	3.33%	**100.00%**	23.33%	**53.33%**	0.00%
LinkedList	**13.33%**	10.00%	**100.00%**	**100.00%**	**20.00%**	3.33%
SensorNetwork	**86.67%**	26.67%	**100.00%**	93.33%	**43.33%**	6.67%
HiPAcc	**100.00%**	6.67%	**100.00%**	**100.00%**	**73.33%**	70.00%
SPLSSimuelESPnP	0.00%	0.00%	**100.00%**	**100.00%**	**23.33%**	6.67%
TankWar	**53.33%**	0.00%	**100.00%**	46.67%	**36.67%**	3.33%
JavaGC	**100.00%**	6.67%	**100.00%**	**100.00%**	43.33%	**60.00%**
Polly	**100.00%**	16.67%	**40.00%**	20.00%	**60.00%**	10.00%
DSSample	0.00%	0.00%	**100.00%**	**100.00%**	0.00%	0.00%
VP9	**100.00%**	10.00%	**100.00%**	60.00%	**33.33%**	3.33%
WebPortal	**3.33%**	**3.33%**	**100.00%**	**100.00%**	**30.00%**	6.67%
JHipster	0.00%	0.00%	**100.00%**	**100.00%**	**30.00%**	23.33%
Drupal	**6.67%**	0.00%	**6.67%**	0.00%	**16.67%**	0.00%
SmartHomev2.2	**6.67%**	0.00%	**100.00%**	76.67%	**10.00%**	6.67%
VideoPlayer	0.00%	0.00%	**10.00%**	0.00%	**16.67%**	0.00%
Amazon	0.00%	**6.67%**	**100.00%**	**100.00%**	6.67%	**20.00%**
ModelTransformation	**100.00%**	3.33%	**100.00%**	60.00%	**13.33%**	3.33%
CocheEcologico	0.00%	0.00%	**100.00%**	**100.00%**	**80.00%**	**80.00%**
n30Model1	**6.67%**	0.00%	**63.33%**	30.00%	**23.33%**	3.33%
n30Model2	**100.00%**	30.00%	**100.00%**	96.67%	**40.00%**	20.00%
n30Model3	**90.00%**	20.00%	**100.00%**	96.67%	**30.00%**	10.00%
n30Model4	**26.67%**	0.00%	**100.00%**	**100.00%**	**76.67%**	13.33%
n30Model5	0.00%	**6.67%**	**100.00%**	60.00%	**73.33%**	3.33%
n30Model6	**100.00%**	0.00%	**100.00%**	93.33%	**46.67%**	6.67%
n30Model7	**80.00%**	3.33%	**100.00%**	**100.00%**	**30.00%**	6.67%
n30Model8	**3.33%**	0.00%	**90.00%**	40.00%	**40.00%**	6.67%
n30Model9	**83.33%**	16.67%	**100.00%**	96.67%	**16.67%**	10.00%
n30Model10	**93.33%**	3.33%	**100.00%**	**100.00%**	**23.33%**	0.00%
n50Model1	0.00%	0.00%	0.00%	0.00%	**33.33%**	3.33%
n50Model2	**10.00%**	0.00%	**100.00%**	**100.00%**	**33.33%**	0.00%
n50Model3	**33.33%**	0.00%	**100.00%**	**100.00%**	**33.33%**	13.33%
n50Model4	**3.33%**	0.00%	**100.00%**	**100.00%**	**33.33%**	10.00%
n50Model5	**100.00%**	20.00%	**100.00%**	**100.00%**	**36.67%**	10.00%
n50Model6	**100.00%**	16.67%	**100.00%**	93.33%	**16.67%**	10.00%
n50Model7	**100.00%**	36.67%	**100.00%**	**100.00%**	**90.00%**	56.67%
n50Model8	**13.33%**	0.00%	**100.00%**	83.33%	**30.00%**	0.00%
n50Model9	**30.00%**	6.67%	**100.00%**	56.67%	**30.00%**	0.00%
n50Model10	**10.00%**	0.00%	**96.67%**	36.67%	**26.67%**	10.00%
Min	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Max	100.00%	36.67%	100.00%	100.00%	100.00%	80.00%
Average	43.21%	7.36%	88.18%	68.49%	40.94%	15.16%



(a) Runtime (measured in seconds) (b) Runtime(MAP-Elites)/Runtime(GrES)

Fig. 16. Runtime comparisons between MAP-Elites and GrES. As an example, we choose YASA as the seeding tool. For the other two seeding tools, similar results are obtained.



(a) Mann-Whitney U test (b) $\widehat{A}_{12}$ statistic

Fig. 17. Summary of theMann-Whitney U test and $\widehat{A}_{12}$ statistic for the comparisons between MAP-Elites and NS regarding test suite diversity and fault detection rate reported in Table 7.

and SamplingCA. Therefore, MAP-Elites is a useful tool in the context of either generating multiple high-performing test suites or reducing the size of covering arrays (i.e., seeds) returned by state-of-the-art t-wise testing tools.

6.4 RQ4: Comparison Regarding Fault Detection Capacity

This section evaluates the fault detection capacity of MAP-Elites and NS [96] on large FMs. Different from the previous sections where MAP-Elites and its competitors are evaluated based on the overall performance with respect to the whole archive, this section evaluates MAP-Elites on the basis of a single test suite in the returned archive. This particular test suite shares the same size with the one returned by NS (recall that NS returns a single test suite in each run). Table 7 gives the results of test suite diversity and fault detection rate for MAP-Elites and NS, and Figure 17 summarizes the Mann-Whitney U test and $\widehat{A}_{12}$ statistic results. Regarding test suite diversity, as can be seen, MAP-Elites significantly outperforms NS on all FMs, except for `busybox-1.18.0` on which the two algorithms obtain similar performance. Moreover, considering the $\widehat{A}_{12}$ statistic, MAP-Elites shows clear improvements over NS, with large differences in favor of MAP-Elites on 98% of the FMs, and with negligible differences on the remaining 2% of the models (i.e., `busybox-1.18.0`).

Regarding the fault detection rate, as can be found from Figure 17(a), MAP-Elites performs better than, equivalently to, and worse than NS on 49%, 43%, and 8% of FMs, respectively. Obviously, the FMs on which MAP-Elites performs better are much more than those on which MAP-Elites

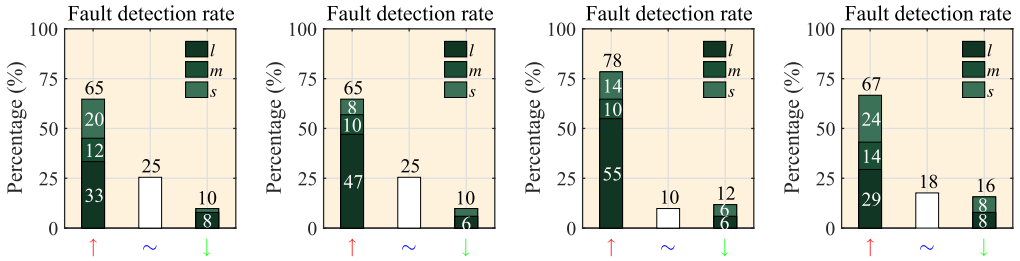
Table 7. Test Suite Diversity and Fault Detection Rate for MAP-Elites and NS on Large FMs

	Test suite diversity				Fault detection rate			
	MAP-Elites	NS	$\widehat{A}_{12}$		MAP-Elites	NS	$\widehat{A}_{12}$	
n100Model1	**6.371e + 01**	6.328e + 01•	1.000	↑ *l*	**9.617e + 01**	9.617e + 01∗	0.464	∼ *n*
n100Model2	**6.382e + 01**	6.373e + 01•	0.967	↑ *l*	**9.742e + 01**	9.742e + 01∗	0.546	∼ *n*
n100Model3	**5.360e + 01**	5.338e + 01•	0.972	↑ *l*	9.850e + 01	**9.883e + 01**○	0.219	↓ *l*
n100Model4	**6.254e + 01**	6.238e + 01•	0.911	↑ *l*	**9.717e + 01**	9.700e + 01∗	0.579	↑ *s*
n100Model5	**6.232e + 01**	6.219e + 01•	0.932	↑ *l*	9.800e + 01	**9.800e + 01**∗	0.450	∼ *n*
n100Model6	**6.209e + 01**	6.171e + 01•	0.818	↑ *l*	9.783e + 01	**9.788e + 01**∗	0.433	∼ *n*
n100Model7	**6.477e + 01**	6.462e + 01•	0.993	↑ *l*	9.765e + 01	**9.777e + 01**∗	0.399	↓ *s*
n100Model8	**5.680e + 01**	5.674e + 01•	0.791	↑ *l*	9.669e + 01	**9.669e + 01**∗	0.487	∼ *n*
n100Model9	**6.156e + 01**	6.150e + 01•	0.874	↑ *l*	9.727e + 01	**9.735e + 01**∗	0.444	∼ *n*
n100Model10	**6.199e + 01**	6.182e + 01•	1.000	↑ *l*	**9.804e + 01**	9.775e + 01∗	0.636	↑ *s*
Printers	**5.869e + 01**	5.855e + 01•	1.000	↑ *l*	9.400e + 01	**9.406e + 01**∗	0.477	∼ *n*
fiasco_17_10	**4.097e + 01**	4.074e + 01•	1.000	↑ *l*	8.185e + 01	**8.306e + 01**○	0.099	↓ *l*
uClibc-ng_1_0_29	**6.144e + 01**	6.107e + 01•	1.000	↑ *l*	**8.122e + 01**	8.098e + 01∗	0.589	↑ *s*
E-shop	**7.007e + 01**	6.973e + 01•	1.000	↑ *l*	**9.620e + 01**	9.588e + 01•	0.774	↑ *l*
toybox	**3.701e + 01**	3.691e + 01•	1.000	↑ *l*	**9.998e + 01**	9.998e + 01∗	0.496	∼ *n*
axTLS	**4.257e + 01**	4.230e + 01•	0.999	↑ *l*	9.967e + 01	**9.967e + 01**∗	0.482	∼ *n*
busybox_1_28_0	**7.575e + 01**	7.559e + 01•	1.000	↑ *l*	**9.384e + 01**	9.354e + 01•	0.823	↑ *l*
SPLOT-FM-1000-1	**6.255e + 01**	6.220e + 01•	1.000	↑ *l*	**8.519e + 01**	8.491e + 01•	0.740	↑ *l*
SPLOT-FM-1000-2	**6.568e + 01**	6.537e + 01•	1.000	↑ *l*	**8.374e + 01**	8.352e + 01•	0.787	↑ *l*
SPLOT-FM-1000-3	**6.438e + 01**	6.408e + 01•	1.000	↑ *l*	**8.161e + 01**	8.148e + 01•	0.659	↑ *s*
SPLOT-FM-1000-4	**6.940e + 01**	6.917e + 01•	1.000	↑ *l*	**8.837e + 01**	8.802e + 01•	0.827	↑ *l*
SPLOT-FM-1000-5	**6.679e + 01**	6.651e + 01•	1.000	↑ *l*	**8.314e + 01**	8.286e + 01•	0.760	↑ *l*
SPLOT-FM-1000-6	**6.847e + 01**	6.818e + 01•	1.000	↑ *l*	**8.600e + 01**	8.567e + 01•	0.819	↑ *l*
SPLOT-FM-1000-7	**6.852e + 01**	6.829e + 01•	1.000	↑ *l*	**8.768e + 01**	8.754e + 01•	0.718	↑ *m*
SPLOT-FM-1000-8	**6.135e + 01**	6.097e + 01•	1.000	↑ *l*	**8.645e + 01**	8.616e + 01•	0.758	↑ *l*
SPLOT-FM-1000-9	**6.309e + 01**	6.281e + 01•	1.000	↑ *l*	**9.046e + 01**	9.023e + 01•	0.871	↑ *l*
SPLOT-FM-1000-10	**6.831e + 01**	6.804e + 01•	1.000	↑ *l*	**8.534e + 01**	8.514e + 01•	0.689	↑ *m*
mpc50	**6.661e + 01**	6.641e + 01•	0.999	↑ *l*	**9.264e + 01**	9.260e + 01∗	0.544	∼ *n*
ref4955	**6.656e + 01**	6.636e + 01•	1.000	↑ *l*	**9.259e + 01**	9.246e + 01∗	0.599	↑ *s*
linux	**6.618e + 01**	6.594e + 01•	1.000	↑ *l*	9.241e + 01	**9.244e + 01**∗	0.543	∼ *n*
csb281	**6.707e + 01**	6.686e + 01•	1.000	↑ *l*	**9.215e + 01**	9.199e + 01•	0.763	↑ *l*
ecos-icse11	**6.713e + 01**	6.690e + 01•	1.000	↑ *l*	**9.211e + 01**	9.200e + 01•	0.761	↑ *l*
ebsa285	**6.702e + 01**	6.679e + 01•	1.000	↑ *l*	**9.193e + 01**	9.178e + 01•	0.741	↑ *l*
vrc4373	**6.674e + 01**	6.651e + 01•	1.000	↑ *l*	**9.242e + 01**	9.224e + 01•	0.693	↑ *m*
pati	**6.665e + 01**	6.643e + 01•	1.000	↑ *l*	**9.276e + 01**	9.258e + 01•	0.804	↑ *l*
dreamcast	**6.648e + 01**	6.626e + 01•	1.000	↑ *l*	**9.265e + 01**	9.251e + 01•	0.694	↑ *m*
pc_i82544	**6.699e + 01**	6.678e + 01•	1.000	↑ *l*	**9.227e + 01**	9.221e + 01∗	0.619	↑ *s*
XSEngine	**6.703e + 01**	6.681e + 01•	1.000	↑ *l*	**9.249e + 01**	9.237e + 01•	0.673	↑ *m*
refidt334	**6.719e + 01**	6.699e + 01•	1.000	↑ *l*	**9.206e + 01**	9.200e + 01∗	0.603	↑ *s*
ocelot	**6.670e + 01**	6.650e + 01•	1.000	↑ *l*	**9.261e + 01**	9.245e + 01•	0.758	↑ *l*
integrator_arm9	**6.660e + 01**	6.637e + 01•	1.000	↑ *l*	**9.214e + 01**	9.211e + 01∗	0.540	∼ *n*
olpcl2294	**6.675e + 01**	6.654e + 01•	1.000	↑ *l*	**9.221e + 01**	9.207e + 01•	0.733	↑ *m*
olpce2294	**6.676e + 01**	6.652e + 01•	1.000	↑ *l*	**9.202e + 01**	9.172e + 01•	0.854	↑ *l*
phycore	**6.660e + 01**	6.640e + 01•	1.000	↑ *l*	**9.267e + 01**	9.248e + 01•	0.744	↑ *l*
hs7729pci	**6.638e + 01**	6.618e + 01•	1.000	↑ *l*	**9.175e + 01**	9.170e + 01∗	0.594	↑ *s*
freebsd-icse11	**7.259e + 01**	7.239e + 01•	1.000	↑ *l*	**9.154e + 01**	9.116e + 01•	0.894	↑ *l*
uClinux	**4.217e + 01**	4.209e + 01•	1.000	↑ *l*	**9.997e + 01**	9.997e + 01∗	0.546	∼ *n*
Automotive01	**5.955e + 01**	5.939e + 01•	1.000	↑ *l*	**8.279e + 01**	8.266e + 01•	0.662	↑ *s*
SPLOT-FM-5000	**6.661e + 01**	6.648e + 01•	1.000	↑ *l*	**8.188e + 01**	8.182e + 01∗	0.606	↑ *s*
busybox-1.18.0	4.531e + 01	**4.531e + 01**∗	0.543	∼ *n*	9.952e + 01	**9.959e + 01**○	0.036	↓ *l*
2.6.28.6-icse11	**7.300e + 01**	7.293e + 01•	1.000	↑ *l*	8.882e + 01	**8.911e + 01**○	0.026	↓ *l*

In NS, the test suite size is 100, and the number of FEs is 2,000. Parameter settings in MAP-Elites follow the specifications given in Section 5.3.

performs worse (i.e., 49% vs 8%). Furthermore, the $\widehat{A}_{12}$ statistic in Figure 17(b) shows that, comparing MAP-Elites with NS, large, medium, and small differences in favor of MAP-Elites vs NS are 33% vs 8%, 12% vs 0%, and 20% vs 2%, respectively. For each magnitude of differences, the percentage in favor of MAP-Elites is much higher than that in favor of NS. Due to limited space, experimental results obtained by using smaller test suite sizes (i.e., 50) and fewer FEs (i.e., 1,000 and 500) are given in the online supplement.[17] Figure 18 shows $\widehat{A}_{12}$ statistics for the fault detection comparisons under different parameter settings. As can be seen, the superiority of MAP-Elites over NS seems to be insensitive to these parameter settings. However, one thing worth noting is that, due to the inherent cooperative search mechanism, MAP-Elites searches for three test suites at a time, and

[17] https://doi.org/10.5281/zenodo.7805017



(a) TS size: 100, FEs: 2000 (b) TS size: 100, FEs: 1000 (c) TS size: 100, FEs: 500 (d) TS size: 50, FEs: 2000

Fig. 18. Summary of the $\widehat{A}_{12}$ statistic for the fault detection rate comparisons between MAP-Elites and NS using different parameter settings.

thus we use tripled FEs. As a natural result, the time cost of MAP-Elites is also approximatively tripled (3.43 times on average according to our experiments). For MAP-Elites, the longest runtime is 467 seconds (≈ 7.8 minutes), which is observed on `2.6.28.6-icse11`.

Now turning to Table 7, it is interesting to observe that MAP-Elites overwhelmingly outperforms NS regarding test suite diversity but shows degeneration regarding fault detection rate on some FMs. For example, MAP-Elites performs significantly better than NS on `2.6.28.6-icse11`, with large differences in its favor regarding test suite diversity. However, the situation regarding fault detection rate is totally different. What is the underlying reason for this phenomenon? Inspired by our previous work [96], we perform correlation analysis to explore how test suite diversity is correlated with fault detection rate. To this end, we generate using PLEDGE [44] 1,000 test suites for each FM, and each test suite is made up of 100 test cases. Subsequently, we calculate test suite diversity and fault detect rate for each test suite. Thus, we get 1,000 pairs of data, based on which a correlation analysis is performed. The correlation analysis returns Pearson's correlation coefficient r and p-values for testing the hypothesis of no correlation between two random variables (null hypothesis). The Pearson's r measures the linear dependence between two variables, and its value ranges from −1 to 1, with −1 representing a perfect negative correlation, 0 representing no correlation and 1 representing a perfect positive correlation. The p-values range from 0 to 1, and values smaller than the significance level (the default is 0.05) indicate that the corresponding correlation is considered significant. The correlation analysis results are provided in Table 8, where we can find that r is positive on the majority of the FMs and negative on only five exceptions: `n100Model3`, `fiasco_17_10`, `uClibc-ng_1_0_29`, `busybox-1.18.0`, and `2.6.28.6-icse11`. Moreover, according to the p values, all positive correlations are statistically significant ($p < 0.05$), except for the one observed on `uClinux` ($p = 0.2398 > 0.05$); in contrast, all negative correlations are statistically insignificant ($p > 0.05$), except for the one observed on `fiasco_17_10`. Turning to Table 7, we can find that positive correlations generally lead to significantly better (or at least statistically equal) fault detection rate of MAP-Elites in comparison with NS. This is because enhancing diversity indirectly improves fault detection rate (due to their positive correlations). Conversely, negative correlations tend to significantly worsen MAP-Elites regarding the fault detection capability, with even large effect sizes against MAP-Elites on four of the preceding five exceptional FM. Nevertheless, since negative correlations are not common, the test suite diversity defined by Equation (3) is still deemed as a useful metric for assessing the potential of a test suite in detecting faults.

Finally, the experiments in this section bring out the following conclusions. First, compared with the recent diversity-driven NS [96] algorithm, MAP-Elites is able to generate significantly more diverse test suites, potentially disclosing more faults on the majority of the FMs under study.

Table 8. Pearson's Correlation Coefficient r and p-Values in the Correlation Analysis between Test Suite Diversity and Fault Detection Rate

	Pearson's r	p
n100Model1	0.3526	$< 10^{-5}$
n100Model2	0.4160	$< 10^{-5}$
n100Model3	−0.0102	0.7484
n100Model4	0.2972	$< 10^{-5}$
n100Model5	0.2885	$< 10^{-5}$
n100Model6	0.2176	$< 10^{-5}$
n100Model7	0.2656	$< 10^{-5}$
n100Model8	0.2482	$< 10^{-5}$
n100Model9	0.0734	0.0202
n100Model10	0.2952	$< 10^{-5}$
Printers	0.1511	$< 10^{-5}$
fiasco_17_10	−0.2409	$< 10^{-5}$
uClibc-ng_1_0_29	−0.0185	0.5592
E-shop	0.3892	$< 10^{-5}$
toybox	0.1945	$< 10^{-5}$
axTLS	0.3742	$< 10^{-5}$
busybox_1_28_0	0.2070	$< 10^{-5}$
SPLOT-FM-1000-1	0.4878	$< 10^{-5}$
SPLOT-FM-1000-2	0.4330	$< 10^{-5}$
SPLOT-FM-1000-3	0.3704	$< 10^{-5}$
SPLOT-FM-1000-4	0.4098	$< 10^{-5}$
SPLOT-FM-1000-5	0.4949	$< 10^{-5}$
SPLOT-FM-1000-6	0.5344	$< 10^{-5}$
SPLOT-FM-1000-7	0.3569	$< 10^{-5}$
SPLOT-FM-1000-8	0.4050	$< 10^{-5}$
SPLOT-FM-1000-9	0.4727	$< 10^{-5}$
SPLOT-FM-1000-10	0.4809	$< 10^{-5}$
mpc50	0.3680	$< 10^{-5}$
ref4955	0.4561	$< 10^{-5}$
linux	0.3740	$< 10^{-5}$
csb281	0.4271	$< 10^{-5}$
ecos-icse11	0.4574	$< 10^{-5}$
ebsa285	0.4155	$< 10^{-5}$
vrc4373	0.4144	$< 10^{-5}$
pati	0.4672	$< 10^{-5}$
dreamcast	0.4568	$< 10^{-5}$
pc_i82544	0.2897	$< 10^{-5}$
XSEngine	0.4163	$< 10^{-5}$
refidt334	0.3945	$< 10^{-5}$
ocelot	0.3836	$< 10^{-5}$
integrator_arm9	0.4526	$< 10^{-5}$
olpcl2294	0.4556	$< 10^{-5}$
olpce2294	0.4946	$< 10^{-5}$
phycore	0.3374	$< 10^{-5}$
hs7729pci	0.4899	$< 10^{-5}$
freebsd-icse11	0.3379	$< 10^{-5}$
uClinux	0.0372	0.2398
Automotive01	0.3540	$< 10^{-5}$
SPLOT-FM-5000	0.5247	$< 10^{-5}$
busybox-1.18.0	−0.0449	0.1561
2.6.28.6-icse11	−0.0157	0.1561

Moreover, the superiority of MAP-Elites seems to be insensitive to parameter settings. Second, the preceding phenomenon can be well explained via correlation analysis, showing that test suite diversity has, in most cases, a significantly positive correlation with the fault detection rate.

6.5 Threats to Validity

This section briefly discusses threats to validity and how they are addressed. We consider internal validity, construct validity, and external validity.

Internal Validity. This concerns any aspects that could result in bias. Threats could arise from the tools used in the experiments, and therefore, where possible, we used tools that have been used in previous experiments (e.g., SamplingCA [67] and PLEDGE [44]). We carefully tested our implementation of MAP-Elites to ensure that it behaves as expected, and the code of the peer algorithms was either written by ourselves (i.e., MI-GA[18]) or taken from standard toolkits (e.g., FeatureIDE,[19] jMetal,[20] TSE_NS[21]). Moreover, all experiments were independently run 30 times to reduce the effect of the stochastic nature of the algorithms and standard statistical tests were used to make reliable comparisons. To evaluate the fault detection capacity, we used the standard fault simulator [2, 6] to generate artificial faults and assumed the presence of an automated oracle to detect faults. This is a common practice in SPL testing (see [29, 85, 96]).

Construct Validity. The objective and measure functions used in the experiments are those that were widely adopted in previous studies. The pairwise coverage ($t = 2$) is a widely used test criterion in SPL testing, and the test suite size is a key factor that should be considered in the decision-making process [61]. In addition, we used the QD-Score since it is the most widely used performance metric for QD optimization; an archive with a high QD-Score provides a set of test suites with good overall performance.

External Validity. This is related to the degree to which we can generalize from the experiments. This kind of threat commonly exists in software engineering research because the whole population of real problems is unknown. We mitigated this threat by benchmarking on 105 FMs with the number of features ranging from 8 to 6,888. Moreover, both realistic and artificial FMs were considered so as to provide a variety of situations.

7 PRACTICAL IMPLICATIONS

In practice, software engineers can adopt our proposed approach in the same way as using a general search-based approach. Our generation approach requires an FM. However, when the available variability model is not an FM, our approach can also be applied if the model can be translated to a Boolean one using transformation rules (e.g., [36]). If such a transformation is unavailable, the proposed approach, combined with Satisfiability Modulo Theory (SMT) solvers, is capable of handling non-Boolean models.

The bounds of the grid (i.e., lb and ub) are parameters of our search-based approach. They jointly determine the number of test suites to return, as well as the interval of their sizes. In fact, these parameters aim at making the test suite generation more flexible, and their specifications entirely depend on practitioners themselves. If the practitioners are interested in test suites with the sizes in a concrete interval, they can set accordingly the preceding two parameters. One thing that practitioners need to be aware of is that more computational resources are generally required if

[18] MI-GA shares most of the code with MAP-Elites. The only difference is that MI-GA uses only test case substitution, and this can be easily implemented by making a minor adaption to Algorithm 3.
[19] https://github.com/FeatureIDE/FeatureIDE
[20] https://github.com/jMetal/jMetal
[21] https://github.com/gzhuxiangyi/TSE_NS

$ub - lb$ is larger. Besides, to adopt our approach in practice, software engineers need to determine the termination condition, being either the amount of time allowed or the number of maximum fitness FEs. Software engineers are free to choose between them, and to set their concrete values. If they have adequate test resources (e.g., time), then they could run the approach longer to get higher-quality test suites.

Similar to multi-objective optimization, the output of our approach is a set of test suites from which practitioners can choose. Based on different test scenarios/test preferences, as illustrated in Section 2.3, practitioners can select a suitable test suite to run on actual systems under test. Technical and economical constraints are the main considerations during this selection process.

Compared with single- and multi-objective approaches, as well as existing t-wise testing tools,[22] the main advantage of the proposed approach is that it can generate multiple and high-performing test suites at a time. However, when using our approach in practice, practitioners should take into account the time to generate seeds. In general, seeds are generated by t-wise testing/sampling algorithms. For SamplingCA, the time to generate seeds significantly varies across FMs. It is quite fast on small FMs, requiring only 0.0046 seconds in the best case. However, it becomes extremely time consuming for large FMs. According to our experiments, the algorithm takes around 4,183 seconds ($\approx$ 1.16 hours) to run to generate a single seed for `2.6.28.6-icse11`. Given the preceding facts, we therefore have the following suggestions for practical use of our approach. If generating seeds is fast (e.g., finishing in seconds), we suggest using seeds; if generating seeds itself is expensive (often on large FMs), we recommend using no seeds. This way, it could significantly reduce the time to generate a test suite. According to Section 6.4, our approach, without seeds, needs only around 7.8 minutes to run on `2.6.28.6-icse11`, far less than 1.16 hours (the time to generate a seed). In the case in which seeds are not used, our approach purely relies on PLEDGE to sample in a loop random and valid test cases, and keeps those that are as diverse as possible. In the context of SPL testing, pursing diversity is a common practice in test case selection/prioritization [2, 42, 96]. Another aspect that should be considered in practice is the choice of the fitness function. For FMs with around 10,000 valid t-sets (e.g., `Amazon`, `ModelTransformation`, and `CocheEcologico` in Table 2), we suggest using t-wise coverage as the fitness function as it is computationally affordable in this case. Otherwise, test suite diversity is recommended because it scales well without the need for checking all valid t-sets during the calculation.

Regarding the execution of test suites, we face an obstacle—source code and unit tests are unavailable[23] for the FMs in Table 2. Therefore, we do not know how long it takes to execute test suites on these SPL systems. Instead, we resort to a recent community-wide dataset [32], which is made up of 30 SPLs with available FM, source code and unit tests.[24] Similar to what we have done in the experiments for RQ3, we run YASA, IncLing, and SamplingCA to generate test suites, then execute them on the real systems. As shown in Figure 19(a), all the three tools are able to generate test suites within 1 second. Regarding the execution of test suites, according to Figure 19(b), the runtime varies across FMs. It is within 1 second on some FMs but can be as large as 1,000+ seconds, which is on observed on `Notepad`. According to our numerical results (provided online), executing test suites is more expensive than generating them on most of FMs.[25]

[22]For existing t-wise testing tools, as in RQ3, it is straightforward to extend the outcome to multiple test suites when test suite size (which is an integer) is used as the behavior descriptor. However, this extension may not be straightforward if other behavior descriptors (e.g., test suite cost [43, 94]) are adopted.

[23]This obstacle is also faced by other research from this community (see, e.g., [5]). Recall that a test case in our context is a product. Following Ferreira et al. [32], only unit tests related to selected features in a product will be executed.

[24]We use 24 out of the 30 SPLs to perform experiments in this work. The excluded SPLs are those that are too small, or those for which we have difficulties in reproducing the experiments due to running environments.

[25]We consider $t = 2$ in the experiments, and the runtime of generating test suites would increase sharply when t increases.

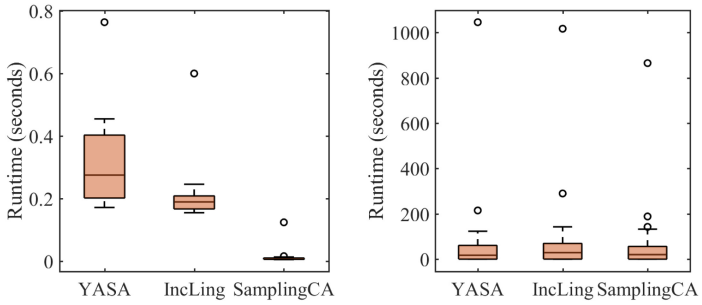


Fig. 19. Runtime to generate and execute test suites for the three t-wise testing tools on the dataset in the work of Ferreira et al. [32].

We then run MAP-Elites, using the same experimental settings as RQ3, to find possibly smaller test suites than seeds of the three tools, and also execute them on the real systems. In this case, the total time of the testing process is the sum of the time spent to generate, optimize, and execute test suites. In the case in which MAP-Elites is not used, the total time is the sum of the time spent to generate and execute test suites. Figure 20 gives the differences between the total time in the preceding two cases. As can be seen, using MAP-Elites makes no difference on FMs like `Chess`, `Email`, and `JTopas` but slightly increases the total time on `FeatureAMP6`, `FeatureAMP7`, and so forth. On the other FMs (e.g., `Companies`, `FeatureAMP1`, and `Notepad`), however, using MAP-Elites is able to significantly save the total time (257.65 seconds at most). A common feature of the preceding FMs is that execution of their test suite is time consuming. Because MAP-Elites successfully reduces the test suite size on the preceding FMs, it saves the time for executing test suites, and the saved time is (much) more than the additional time that MAP-Elites takes. As a result, the total time is reduced. In practice, our approach would be particularly useful in testing SPLs for which executing the test suite is expensive.[26] In fact, evidence shows that such real SPLs commonly exist in practice. For example, for the UAV (Unmanned Aerial Vehicle) product line [5, 69], executing the 2-wise covering array with 22 products lasted an estimated 848 hours [69]. Roughly, each test case/product runs 38.5 hours on average. In this case study, our approach (using MAP-Elites) successfully reduces the size of the 2-wise covering array of YASA, IncLing, and SamplingCA from 23, 28, and 22 to 20, 23, and 20, respectively. This potentially saves 13.04% (2/23), 17.86% (5/28), and 9.09% (2/22) of the total execution time of the test suite—that is, 115.5, 192.5, and 77.0 hours regarding the estimated time.[27] However, our approach additionally takes only around 33 seconds to reduce the test suites, which could be negligible compared to the saved execution time.

8 RELATED WORK

Related work reviewed in this article covers both search-based SPL testing (Section 8.1) and applications of QD optimization (Section 8.2).

[26]When optimizing test suites is much more time consuming than executing them (e.g., the Checkstyle case study where the execution takes only 10+ seconds but the optimization takes 1,400+ seconds (data are provided online)), the effect of our approach would be discounted.

[27]Because the UAV involves complex MATLAB/Simulink, we did not run actually the test suite but referred to the work of Markiegi et al. [69] for estimated execution time. In fact, citing the estimated time aims at showing that running the test suite is expensive in this cast study.

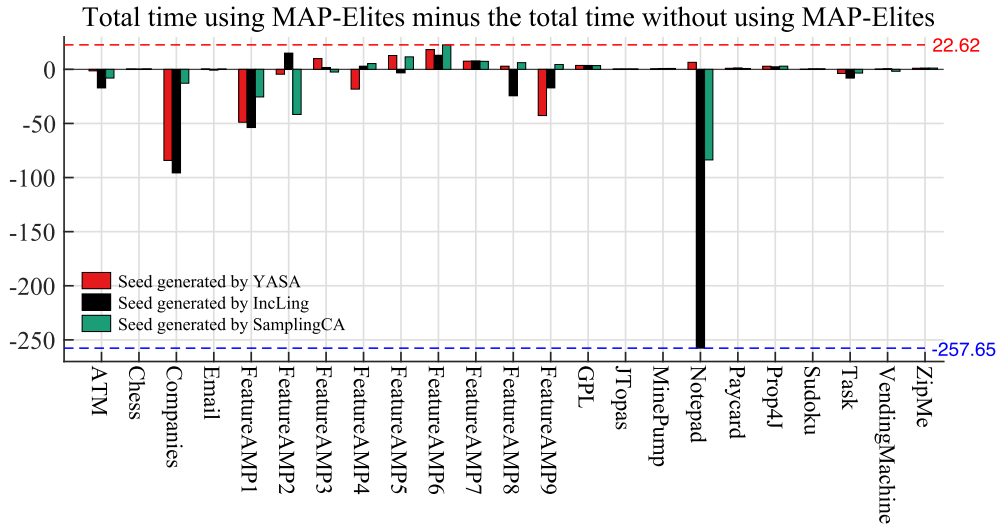


Fig. 20. The differences between the total time with and without using MAP-Elites.

8.1 Search-Based Testing for SPLs

In this section, we briefly summarize related work on search-based testing for SPLs using both single- and multi-objective approaches (Table 9).

8.1.1 Single-Objective Approaches. A majority of the single-objective testing techniques for SPLs are based on feature combination coverage using CIT (combinatorial interaction testing). Systematic mapping studies [25, 52, 62] revealed that most of the papers derive test cases from variability models (typically an FM) using t-wise testing, in which the goal is to find a minimal test suite that cover all possible combinations among t features by at least one test case. Put it another way, the fitness function in this testing technique is t-wise coverage, mostly pairwise coverage (i.e., $t = 2$). However, achieving full t-wise coverage is NP-hard. Hence, genetic and greedy algorithms were often applied to generate test suites covering t-wise feature combinations as many as possible. Representative t-wise testing algorithms include Chvatal [15, 46], ICPL [47], IncLing [1], CASA [37], YASA [50], LS-Sampling [66], and SamplingCA [67]. However, according to other works [32, 70, 78], most of the t-wise algorithms do not scale to large SPLs and/or high interaction strengths. Focusing on pairwise testing, it should be noted that the recent SamplingCA [67] scales well to large SPLs, capable of generating much smaller pairwise covering arrays, and at the same time running one to two orders of magnitude faster than state-of-the-art t-wise algorithms. Given its high effectiveness and efficiency, we choose it as the seeding algorithm in our experiments.

Ensan et al. [29] proposed an evolutionary test suite generation approach for SPLs with a suitable tradeoff balance between fault detection capability and feature coverage. The fitness of a test suite was measured by the combination of two metrics: variability coverage (VC) and cyclomatic complexity (CC). Inspired by their work, the two metrics were adapted and used by Hierons et al. [45] in their work for many-objective SPL test suite generation. In a related paper, Bagheri et al. [6] provided eight coverage criteria based on the transformation of FMs into formal context-free grammars. Experiments performed on several SPLOT FMs show that the test suite generation strategies based on the proposed coverage criteria are effective in significantly reducing the number of required test cases and at the same time preserving an acceptable fault coverage.

Table 9. Summary of Representative Works on Search-Based SPL Testing

Work	Year	Obj. (#)	Objective function(s)
[47]	2012	1	t-wise coverage
[37]	2011	1	t-wise coverage
[1]	2016	1	2-wise coverage
[50]	2020	1	t-wise coverage
[66]	2021	1	2-wise coverage
[67]	2022	1	2-wise coverage
[29]	2012	1	Combination of VC and CC
[98]	2013	1	Code (e.g., branch) coverage
[41]	2014	1	Mutation score
[42]	2014	1	Test suite similarity
[2]	2019	1	Test suite similarity
[96]	2022	1	Test suite diversity
[43]	2013	3	Pairwise coverage, number of test cases, test cost
[95]	2014	3	Test minimization percentage, pairwise coverage, fault detection capability
[94]	2015	5	The preceding three functions plus average execution frequency, overall execution time
[60]	2014	2	Pairwise coverage, test suite size
[24]	2016	2	Distance among products, distance among test cases
[33, 89]	2016, 2017	4	Number of products, mutation score, pairwise coverage, similarity between the products
[77]	2016	7	Pairwise coverage, dissimilarity, number of changes, coefficient of connectivity-density, etc.
[69]	2017	3	Fault detection capability, test execution time, test case appearance frequency
[45]	2020	9	Pairwise coverage, test suite size, test suite cost, etc.
[34]	2020	7	Number of products, alive mutants, uncovered pairs, products similarity, etc.

Xu et al. [98] proposed an approach, called *CONTESA*, for generating test cases for SPLs using test suite augmentation. The tool first selects a single product in a product family and then iteratively selects the next product for testing, generating new test cases for that product to cover only the code (e.g., branches) that are not covered by previous test cases. For each product to be tested, they used a genetic algorithm to automatically generate test cases.

Henard et al. [41] proposed a mutation-based test suite generation for SPLs based on the observation that mutation, as a promising alternative to the CIT criterion, also correlates with fault detection [76]. Their approach starts by creating a set of defective versions of the FM, so-called mutants. Then, the (1+1) Evolutionary Algorithm (or (1+1) EA) is used to search for the test suite with the maximum *mutation score*, which is evaluated based on the number of killed mutants. Note that a mutant is killed if it is not satisfied by at least one of the members in that test suite.

Henard et al. [42] presented a search-based approach to generate and prioritize t-wise test suites. In that approach, they employed an adaptation of the (1+1) EA with no crossover to optimize the fitness function defined based on Jaccard's similarity distance instead of the t-wise coverage. This makes the search-based approach scalable and capable of generating test suites for large SPLs and high t-wise strengths ($t \in [2, 6]$). The underlying assumption of their work is that the higher the distances between the test cases in the suite, the higher the t-wise coverage. In addition, the similarity distances are also used for prioritizing the test suite. One of the feasible ways is to select at each step the test case which is the most distant to all the ones already selected during

the previous steps. Later, Al-Hajjaji et al. [2] proposed a similar similarity-based SPL test suite prioritization method in which an adaptation of the Hamming distance is used to measure the similarity among test cases.

Our previous work [96] generalized the fitness function of Henard et al. [42] by introducing the concept of novelty score [54]. In our previous work [96], the fitness of a test suite is the sum of novelty scores over all members in the test suite. The novelty score of a single test case is defined as the average distance to its k nearest neighbors in the suite. Different from both Henard et al. [42] and Al-Hajjaji et al. [2], Anti-dice distance [24] is used to measure similarity among test cases due to its good performance observed on real SPLs [96]. It should be noted that although the new fitness function was first presented in our previous work [96], it was not explicitly used as an objective there. This article takes a step further in exploiting the benefits of this fitness function in the context of SPL test suite generation based on QD optimization.

8.1.2 Multi-Objective Approaches. Henard et al. [43] presented a multi-objective genetic algorithm for test case generation from SPLs. They considered three objectives: maximizing pairwise coverage, minimizing number of test cases, and minimizing test cost, and combined them into a single fitness function with weights. Experiments conducted on eight FMs with up to 94 features demonstrated the effectiveness, feasibility, and practicality of their approach.

Wang et al. [95] introduced a search-based multi-objective test prioritization technique for SPLs, considering four optimization objectives: minimization of execution cost, maximization of number of prioritized test cases, feature pairwise coverage, and fault detection capability. A fitness function was defined by weighting these objectives, and was adopted in four search-based algorithms, namely an alternative variable method, a custom genetic algorithm, (1 + 1) EA, and random search. The experiments were conducted on Cisco's industrial case study with 53 features and 500 randomly generated models with up to 500 features. The results showed that (1 + 1) EA achieved the best performance for solving the test case prioritization problem.

Wang et al. [93] proposed an approach to minimize the test suite for testing a single product of an SPL while preserving fault detection capability and testing coverage of the original test suite. They formally defined three objectives (i.e., test minimization percentage, pairwise coverage, and fault detection capability) and combined them into one fitness function using weights to guide three different weight-based genetic algorithms. Their approaches were evaluated on four FMs from an industrial project plus five FMs from the SPLOT repository with the number of features ranging from 17 to 77. In a subsequent study [94], the authors compared their weight-based genetic algorithms with seven multi-objective search algorithms, namely NSGA-II, MOCell, SPEA2, PAES, SMPSO, CellDE, and random search, using an industrial case study and 500 artificial problems inspired from that industrial case study. To guide the search, they considered five objectives, four of which measure effectiveness and the remaining one measures cost. Their results indicated that Random-Weighted GA, in which the preceding five objectives are combined into a scalar one using dynamic weights, is promising for the test suite minimization in the context of SPL testing.

Lopez-Herrejon et al. [59, 60] modeled the SPL test suite generation as a multi-objective optimization problem where maximizing pairwise coverage and minimizing test suite size were deemed equally important. In another work, Lopez-Herrejon et al. [60] compared four classical multi-objective algorithms (NSGA-II, MOCell, SPEA2, and PAES), finding that there does not exist a clear winner. Moreover, they studied the impact of three different seeding strategies on computing test suites with maximum coverage and minimum size. These results suggest that the seeding strategy does have an impact on the performance of the search algorithms. In a subsequent study [61], they provided an overview of the state of the art in SPL testing with evolutionary techniques. It should be mentioned that, different from their work [59, 60], we model in this article the SPL test

suite generation as a QD optimization problem with pairwise coverage being the objective and test suite size being the behavior. Since pairwise coverage and test suite size are used in both the work of Lopez-Herrejon et al. [60] and ours, we performed an experiment to compare MAP-Elites with NSGA-II, which solves the multi-objective formalization [60]. Our results show that MAP-Elites is more effective than NSGA-II in generating a large set of diverse and high-performing test suites.

Lopez-Herrejon et al. [63] suggested a parallel genetic algorithm for the generation of a prioritized pairwise SPL test suite with minimum size. Their algorithm followed the master-slave model to parallelize the evaluation of the test suites. They performed an extensive and comprehensive analysis of the algorithm using 235 FMs with different characteristics. Later, Ferrer et al. [35] proposed two hybrid algorithms using integer programming to handle also the prioritized pairwise test suite generation problem. Their study revealed that the hybrid algorithms outperformed the parallel genetic algorithm in terms of both solution quality and computation time.

Devroey et al. [24] proposed a dissimilarity-driven bi-objective test case generation algorithm maximizing the distance among products (product coverage), and the distance among test cases regarding behavioral actions, which are derived from the Featured Transition System representing behavioral aspects of an SPL. The two distances/dissimilarities are combined into a single fitness function to be used in a (1+1) EA. Their approach was evaluated on four case studies with up to 44 features.

Ferreira et al. [33] and Strickler et al. [89] introduced a hyperheuristic (HH) approach to dynamically select the best evolutionary operators during the execution of MOEAs. They implemented the approach in four MOEAs: NSGA-II, SPEA2, IBEA, and MOEA/D-DRA, and applied them to the SPL test suite generation problem with four objectives: minimizing the number of products, maximizing mutation score, maximizing pairwise coverage, and maximizing similarity between the products. The experiments were performed on four realistic FMs with sizes ranging from 14 to 22.

Parejo et al. [77] presented a case study on multi-objective test case prioritization in the real-world Drupal web framework. They designed seven prioritization objective functions based on functional and non-functional properties of the SPL under test. The authors performed several experiments using NSGA-II to compare the effectiveness of 63 different combinations of up to three of these objectives in accelerating the detection of faults in Drupal. The results showed that multi-objective prioritization generally performs better than mono-objective prioritization, and that the pairwise coverage combined with other objectives is usually effective in detecting bugs quickly.

Markiegi et al. [69] proposed a search-based approach to test SPLs by allocating a small number of test cases for each of the products to be tested. To guide the search, a fitness function was designed by combining three objectives: maximizing fault detection capability, minimizing test execution time, and maximizing test case appearance frequency. Their approach was evaluated using a cyber-physical system SPL with 46 features.

Recently, Hierons et al. [45] extended the preceding work by modeling SPL test suite generation as a many-objective optimization problem, considering nine state-of-the-art objective functions. Three functions (pairwise coverage, test suite size, and test suite cost) are related to test case selection (i.e., selecting test cases to be tested), and six functions address test case prioritization (i.e., arranging products in an order such that faults are detected as quickly as possible). Based on domain knowledge, they devised a novel approach, called *GrES*, to solve the preceding optimization problem. In GrES, all nine objectives are considered, but the pairwise coverage is seen as the most important and thus is given the highest priority. This is inspired by the observation that typically, when testing an SPL, the testers will want to guarantee that all pairs of features are tested. The authors evaluated GrES on both randomly generated and realistic FMs with up to 500 features, and their results of evaluations showed that this approach

is promising, outperforming in general previously proposed techniques (e.g., PLEDGE [44]) and several many-objective optimization algorithms (e.g., MOEA/D [99] and SPEA2+SDE [57]).

Similarly, Ferreira et al. [34] also formulated the test case selection for variability testing of SPLs as a many-objective optimization problem, considering seven objectives identified in previous works related to SPL testing. To support the test case selection, they developed a tool called *Nautilus/VTSPL*, which allows the testers to choose (or configure) the objectives of interest and to apply different categories of multi-/many-objective algorithms, including dimensionality reduction and preference-based algorithms with (or without) human participation. Moreover, this tool also offers support to visualization of the generated products to ease the decision-making process.

In the preceding work, the SPL test suite generation was modeled as a multi-/many-objective problem which was then solved by either transforming it into a single-objective problem with weighting factors (see, e.g., [43, 93, 95]) or directly applying MOEAs (see, e.g., [33, 34, 45, 60, 77, 89]). Different from all the preceding work, this work provides a new mathematical model for the test suite generation problem and applies a new paradigm of evolutionary algorithms (i.e., QD optimization).

8.2 Applications of QD Optimization

In this section, we review typical applications of QD optimization, particularly to search-based software testing. In fact, QD optimization algorithms have been successfully employed in many domains, especially evolutionary robotics and video games. In robotics, for example, it has been used to produce behavioral repertoires that enable robots to quickly adapt to damage [18], perform complex tasks [27], and walk in every direction [21]; morphological designs for robotic organisms [55] as well as walking "soft robots" [72]; and controllers that drive simulated robots through mazes [54, 81]. In video games, QD algorithms have been widely applied to procedural content generation [38, 58] and game playing [12, 28].

Other than the preceding domains, there have been several works on the application of QD algorithms to SBSE [40]. Romero et al. [83] recently pointed out that NS seems to well fit precepts of SBSE because it emphasizes novel individuals that were not found previously. This is in line with the goal of SBSE—that is, searching for real but unpredictable solutions [83]. Currently, the NS algorithm has been applied to the next release problem [83], software improvement [65], and automatic bug repair [92].

Regarding software testing, Boussaa et al. [11] applied, for the first time, NS to the test data generation problem. They believe that NS-based test data generation is attractive because it allows the exploration of the huge space of test data within the input domain. Marculescu et al. [68] performed an experiment comparing exploration-based algorithms (including NS and MAP-Elites) with an objective-based one for testing a clustering algorithm. Their conclusion is that exploration-focused algorithms are useful in investigating high-dimensional behavior spaces, even in the cases in which limited information and limited resources are available. Taking inspiration from QD algorithms, Feldt and Poulding [30] explored different approaches to generate test inputs with high feature-specific diversity. Some of the approaches they investigated are based on the general strategies employed in NS and MAP-Elites.

Besides traditional software, QD algorithms have also been used to test **Deep Learning (DL)** systems. Riccio and Tonella [82] developed a search-based tool called *DeepJanus* to generate a frontier of behaviors for a DL system, which is a set of pairs of inputs that are similar to each other but trigger different behaviors of the system. The frontier provides the developers with useful information to assess the quality of a DL system and to identify valid inputs that cannot be handled properly. To achieve a thorough exploration of the frontier of behaviors, DeepJanus hybridizes a traditional multi-objective search-based algorithm (i.e., NSGA-II) with NS. This is achieved by

defining a fitness function that includes a measure of sparseness of the solutions (i.e., novelty score introduced in NS). The authors declared that if the sparseness was not included in the fitness function, then the search tended to get stuck in local optima, covering only a tiny part of the frontier.

Zohdinasab et al. [101] presented the tool DeepHyperion, an open source automated test input generator for DL systems that leverages the key advantage of QD algorithms. Specifically, DeepHyperion uses MAP-Elites to automatically generate a large and diverse set of high-performing (i.e., misbehaving or near-misbehaving) test inputs. With the tool, the developers are allowed to interpret the inputs that trigger a misbehavior in terms of their feature values. To define the feature space of interest, the authors also proposed a systematic methodology, which can be used in conjunction with DeepHyperion, to support the identification and quantification of features that well characterize the generated inputs. It has been shown by the evaluations on a classification problem (handwritten digit recognition) and a regression problem (steering angle prediction in a self-driving car) that DeepHyperion is effective in generating diverse failure-inducing inputs for the corresponding DL systems under test.

In related work [102], the same authors enhanced DeepHyperion, leading to a more efficient and effective DL system test generator, DeepHyperion-CS. The main extension is that instead of performing a random selection, DeepHyperion-CS chooses with higher probability the inputs with larger *contribution score*, a novel metric to select from the archive the candidate inputs (to be mutated) that are more likely to increase the exploration of the feature space. The empirical study on the same test subjects as in the work of Zohdinasab et al. [101] showed that DeepHyperion-CS significantly improves the efficiency and effectiveness of DeepHyperion in finding misbehavior-inducing inputs and exploring the feature space.

Results in other works [11, 68, 82, 101, 102] suggested that QD optimization is a new and promising tool for search-based software testing. In the context of SPL testing, our previous work [96] has adopted NS to generate a single test suite with diverse test cases. In this work, we take a step further by leveraging a more advanced QD algorithm (i.e., MAP-Elites) to generate a set of diverse test suites at a time, easing the decision-making process.

9 CONCLUSION AND FUTURE WORK

In the practice of SPL testing, software engineers may need to analyze a lot of test suites and, in the end, choose the best one to use according to different test scenarios or their own preferences [61]. To achieve this, it is required that a large set of diverse and high-performing test suites are generated, and this is exactly the goal in QD optimization [80]. Motivated by the preceding, we model, for the first time, the test suite generation for SPLs as a QD optimization problem. This is a totally new perspective to build the mathematical model for SPL testing. It should be noted that the multi-objective model also generates a solution set, but it generally requires the conflicts of the objective functions. In contrast, the QD model does not impose such a requirement and thus is more generic and flexible.

The QD optimization is characterized by an objective function to optimize and a behavior space to span. The behavior space is often tailored by users based on their own test scenarios/preferences. The customization of this space allows users to express the types of solutions that they are interested in. In this work, we choose test suite size as the behavior because it is an important feature to characterize a test suite. Regarding the objective function, we define it as the t-wise coverage (precisely pairwise coverage) for small FMs and test suite diversity for large ones. As shown in our previous work [96], test suite diversity has significantly positive correlations with t-wise coverage, and at the same time, it scales well with respect to the size of FMs. Therefore, our model can handle both small and large FMs (the largest one in our experiments has 6,888 features).

We adopt MAP-Elites, one of the most widely used QD algorithms to solve the model. MAP-Elites discretizes the behavior space into grid and searches for the best possible solution for each of the cells in that grid. In this context, each cell corresponds to a QD subproblem. It is shown that MAP-Elites outperforms the single-objective algorithm (with an independent run for each QD subproblem) regarding the overall performance (measured by QD-Score) of the returned solution set. We examine the underlying reasons, finding that the superiority of MAP-Elites over the single-objective approach is due to its inherent mechanism enabling information sharing among QD subproblems. Moreover, by using pairwise coverage and test suite size as two objectives, the test suite generation is reformulated as a bi-objective optimization problem, which is solved by NSGA-II, the most widely used MOEA in SBSE. It is also shown that MAP-Elites outperforms NSGA-II; we find out the reason—constrained competition in MAP-Elites is more powerful than the global competition in NSGA-II in maintaining diversity of the solution set.

In the experiments, MAP-Elites is also compared with three existing t-wise testing tools: YASA [50], IncLing [1], and SamplingCA [67]. Our results indicate that MAP-Elites outperforms two of them regarding generating a set of high-performing test suites. Moreover, MAP-Elites, which performs better than GrES, can be used as an effective tool for reducing the size of the covering arrays returned by these t-wise tools. Finally, we compare MAP-Elites against a recent NS-based test suite generation approach. Results show that test suites of MAP-Elites are more diverse than those of its competitor, and interestingly, they tend to expose more faults. This phenomenon is explained by our correlation analysis, revealing the positive correlations between test suite diversity and fault detection rate in most cases. We make our code and data publicly available for follow-up studies on GitHub (https://github.com/gzhuxiangyi/SPLTestingMAP) and Zenodo (https://doi.org/10.5281/zenodo.7805017).

This work takes the first step in exploiting the benefits of QD optimization and its algorithm in the context of SPL testing. There are a number of possible lines for future studies. First, the behavior space can be naturally extended to two or more dimensions by incorporating other behaviors, such as test suite cost [43, 94]. Moreover, the test suite generation could also be modeled as a multi-objective QD optimization problem [79] by considering two or more fitness functions. Like in the work of Hierons et al. [45], these fitness functions can concern both test case selection (i.e., selecting a subset of products to test) and test case prioritization (i.e., ordering products to detect bugs as soon as possible). Second, there is value in using more advanced QD algorithms [20, 87], and utilizing more advanced SAT solvers [67, 97] to generate new test cases in the mutation operation. Finally, it is of practical significance to evaluate our test suite generation approach regarding the effectiveness in identifying real faults on a community-wide dataset of real-world SPLs [32].

REFERENCES

[1] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient product-line testing using incremental pairwise sampling. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'16)*. ACM, New York, NY, 144–155.

[2] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective product-line testing using similarity-based product prioritization. *Software & Systems Modeling* 18 (2019), 499–521.

[3] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, 1–10.

[4] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[5] Aitor Arrieta, Sergio Segura, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. 2018. Spectrum-based fault localization in software product lines. *Information and Software Technology* 100 (2018), 18–31. DOI: http://dx.doi.org/https://doi.org/10.1016/j.infsof.2018.03.008

[6] Ebrahim Bagheri, Faezeh Ensan, and Dragan Gasevic. 2012. Grammar-based test generation for software product line feature models. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research (CASCON'12)*. 87–101.

[7] Eduard Baranov, Axel Legay, and Kuldeep S. Meel. 2020. Baital: An adaptive weighted sampling approach for improved t-wise coverage. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. ACM, New York, NY, 1114–1126.

[8] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference Software Product Lines (SPLC'05)*. 7–20.

[9] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636.

[10] Daniel Le Berre and Anne Parrain. 2010. The Sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), 59–64.

[11] Mohamed Boussaa, Olivier Barais, Gerson Sunyé, and Benoît Baudry. 2015. A novelty search approach for automatic test data generation. In *Proceedings of the 8th International Workshop on Search-Based Software Testing (SBST'15)*. IEEE, Los Alamitos, CA, 40–43.

[12] Rodrigo Canaan, Julian Togelius, Andy Nealen, and Stefan Menzel. 2019. Diverse agents for ad-hoc cooperation in Hanabi. In *Proceedings of the 2019 IEEE Conference on Games (CoG'19)*. 1–8. DOI: http://dx.doi.org/10.1109/CIG.2019.8847944

[13] Konstantinos Chatzilygeroudis, Antoine Cully, Vassilis Vassiliades, and Jean-Baptiste Mouret. 2021. Quality-diversity optimization: A novel branch of stochastic optimization. In *Black Box Optimization, Machine Learning, and No-Free Lunch Theorems*, Panos M. Pardalos, Varvara Rasskazova, and Michael N. Vrahatis (Eds.). Springer, 109–135. DOI: http://dx.doi.org/10.1007/978-3-030-66515-9

[14] Konstantinos Chatzilygeroudis, Antoine Cully, Vassilis Vassiliades, and Jean-Baptiste Mouret. 2021. Quality-diversity optimization: A novel branch of stochastic optimization. In *Black Box Optimization, Machine Learning, and No-Free Lunch Theorems*. Springer, 109–135.

[15] Vaclav Chvatal. 1979. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research* 4, 3 (1979), 233–235.

[16] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman.

[17] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2007. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*. ACM, New York, NY, 129–139.

[18] Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. 2015. Robots that can adapt like animals. *Nature* 521, 7553 (2015), 503–507.

[19] Antoine Cully and Yiannis Demiris. 2018. Hierarchical behavioral repertoires with unsupervised descriptors. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'18)*. ACM, New York, NY, 69–76. DOI: http://dx.doi.org/10.1145/3205455.3205571

[20] Antoine Cully and Yiannis Demiris. 2018. Quality and diversity optimization: A unifying modular framework. *IEEE Transactions on Evolutionary Computation* 22, 2 (2018), 245–259. DOI: http://dx.doi.org/10.1109/TEVC.2017.2704781

[21] A. Cully and J.-B. Mouret. 2016. Evolving a behavioral repertoire for a walking robot. *Evolutionary Computation* 24, 1 (2016), 59–88. DOI: http://dx.doi.org/10.1162/EVCO_a_00143

[22] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2011. A systematic mapping study of software product lines testing. *Information and Software Technology* 53, 5 (2011), 407–423. DOI: https://doi.org/10.1016/j.infsof.2010.12.003

[23] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and Tamt Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[24] Xavier Devroey, Gilles Perrouin, Axel Legay, "P.-Y."Schobbens, and Patrick Heymans. 2016. Search-based similarity-driven behavioural SPL testing. In *Proceedings of the 10th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'16)*. ACM, New York, NY, 89–96. DOI: http://dx.doi.org/10.1145/2866614.2866627

[25] Ivan do Carmo Machado, John D. McGregor, Yguarata Cerqueira Cavalcanti, and Eduardo Santana de Almeida. 2014. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology* 56, 10 (2014), 1183–1199. DOI: https://doi.org/10.1016/j.infsof.2014.04.002

[26] Stephane Doncieux, Alban Laflaquière, and Alexandre Coninx. 2019. Novelty search: A theoretical perspective. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'19)*. ACM, New York, NY, 99–106.

[27] Miguel Duarte, Jorge Gomes, Sancho Moura Oliveira, and Anders Lyhne Christensen. 2018. Evolution of repertoire-based control for robots with complex locomotor systems. *IEEE Transactions on Evolutionary Computation* 22, 2 (2018), 314–328. DOI: http://dx.doi.org/10.1109/TEVC.2017.2722101

[28] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. 2021. First return, then explore. *Nature* 590 (2021), 580–586.

[29] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gašević. 2012. Evolutionary search-based test generation for software product line feature models. In *Advanced Information Systems Engineering*. Lecture Notes in Computer Science, Vol. 7328. Springer, 613–628.

[30] Robert Feldt and Simon M. Poulding. 2017. Searching for test data with feature diversity. *CoRR abs/1709.06017* (2017).

[31] Fischer Ferreira, João P. Diniz, Cleiton Silva, and Eduardo Figueiredo. 2019. Testing tools for configurable software systems: A review-based empirical study. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'19)*. ACM, New York, NY, Article 6, 10 pages. DOI: http://dx.doi.org/10.1145/3302333.3302344

[32] Fischer Ferreira, Gustavo Vale, Joao P. Diniz, and Eduardo Figueiredo. 2021. Evaluating t-wise testing strategies in a community-wide dataset of configurable software systems. *Journal of Systems and Software* 179 (2021), 110990. DOI: https://doi.org/10.1016/j.jss.2021.110990

[33] Thiago N. Ferreira, Jackson A. Prado Lima, Andrei Strickler, Josiel N. Kuk, Silvia R. Vergilio, and Aurora Pozo. 2017. Hyper-heuristic based product selection for software product line testing. *IEEE Computational Intelligence Magazine* 12, 2 (2017), 34–45. DOI: http://dx.doi.org/10.1109/MCI.2017.2670461

[34] Thiago Nascimento Ferreira, Silvia Regina Vergilio, and Mauroane Kessentini. 2020. Many-objective search-based selection of software product line test products with Nautilus. In *Proceedings of the 24th ACM International Systems and Software Product Line Conference—Volume B (SPLC'20)*. ACM, New York, NY, 1–4. DOI: http://dx.doi.org/10.1145/3382026.3431248

[35] Javier Ferrer, Francisco Chicano, and Enrique Alba. 2017. Hybrid algorithms based on integer programming for the search of prioritized test data in software product lines. In *Applications of Evolutionary Computation*, Giovanni Squillero and Kevin Sim (Eds.). Springer International, Cham, Switzerland, 3–19.

[36] Alan M. Frisch and Timothy J. Peugniez. 2001. Solving non-Boolean satisfiability problems with stochastic local search. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence—Volume 1 (IJCAI'01)*. 282–288.

[37] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering* 16, 1 (2011), 61–102.

[38] Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. 2019. Procedural content generation through quality diversity. In *Proceedings of the 2019 IEEE Conference on Games (CoG'19)*. 1–8. DOI: http://dx.doi.org/10.1109/CIG.2019.8848053

[39] Luca Grillotti and Antoine Cully. 2022. Unsupervised behavior discovery with quality-diversity optimization. *IEEE Transactions on Evolutionary Computation* 26, 6 (2022), 1539–1552. DOI: http://dx.doi.org/10.1109/TEVC.2022.3159855

[40] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys* 45, 1 (Dec. 2012), Article 11, 61 pages. DOI: http://dx.doi.org/10.1145/2379776.2379787

[41] Christopher Henard, Mike Papadakis, and Yves Le Traon. 2014. Mutation-based generation of software product line test configurations. In *Proceedings of the 6th International Symposium on Search-Based Software Engineering (SSBSE'14)*. 92–106. DOI: http://dx.doi.org/10.1007/978-3-319-09940-8_7

[42] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering* 40, 7 (July 2014), 650–670.

[43] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. Multi-objective test generation for software product lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*. ACM, New York, NY, 62–71. DOI: http://dx.doi.org/10.1145/2491627.2491635

[44] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. PLEDGE: A product line editor and test generation tool. In *Proceedings of the 17th International Software Product Line Conference Co-Located Workshops (SPLC'13 Workshops)*. ACM, New York, NY, 126–129.

[45] Robert M. Hierons, Miqing Li, Xiaohui Liu, Jose Antonio Parejo, Sergio Segura, and Xin Yao. 2020. Many-objective test suite generation for software product lines. *ACM Transactions on Software Engineering and Methodology* 29, 1 (Jan. 2020), Article 2, 46 pages. DOI: http://dx.doi.org/10.1145/3361146

[46] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of realistic feature models make combinatorial testing of product lines feasible. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS'11)*. 638–652.

[47] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference—Volume 1 (SPLC'12)*. ACM, New York, NY, 46–55.

[48] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference—Volume 1*. ACM, New York, NY, 46–55.

[49] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is there a mismatch between real-world feature models and product-line research? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, New York, NY, 291–302. DOI: http://dx.doi.org/10.1145/3106237.3106252

[50] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: Yet another sampling algorithm. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'20)*. ACM, New York, NY, Article 4, 10 pages.

[51] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. 2004. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421.

[52] Jihyun Lee, Sungwon Kang, and Pilsu Jung. 2020. Test coverage criteria for software product line testing: Systematic literature review. *Information and Software Technology* 122 (2020), 106272. DOI: https://doi.org/10.1016/j.infsof.2020.106272

[53] Jihyun Lee, Sungwon Kang, and Danhyung Lee. 2012. A survey on software product line testing. In *Proceedings of the 16th International Software Product Line Conference—Volume 1 (SPLC'12)*. ACM, New York, NY, 31–40. DOI: http://dx.doi.org/10.1145/2362536.2362545

[54] Joel Lehman and O. Stanley Kenneth. 2011. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation* 19, 2 (2011), 189–223.

[55] Joel Lehman and Kenneth O. Stanley. 2011. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO'11)*. ACM, New York, NY, 211–218. DOI: http://dx.doi.org/10.1145/2001576.2001606

[56] Bingdong Li, Jinlong Li, Ke Tang, and Xin Yao. 2015. Many-objective evolutionary algorithms: A survey. *ACM Computing Surveys* 48, 1 (Sept. 2015), 1–35.

[57] Miqing Li, Shengxiang Yang, and Xiaohui Liu. 2014. Shift-based density estimation for Pareto-based algorithms in many-objective optimization. *IEEE Transactions on Evolutionary Computation* 18, 3 (June 2014), 348–365.

[58] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. 2015. Constrained novelty search: A study on game content generation. *Evolutionary Computation* 23, 1 (2015), 101–129. DOI: http://dx.doi.org/10.1162/EVCO_a_00123

[59] Roberto E. Lopez-Herrejon, Francisco Chicano, Javier Ferrer, Alexander Egyed, and Enrique Alba. 2013. Multi-objective optimal test suite computation for software product line pairwise testing. In *Proceedings of the IEEE International Conference on Software Maintenance*. 404–407.

[60] Roberto E. Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Alexander Egyed, and Enrique Alba. 2014. Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of software product lines. In *Proceedings of the 2014 IEEE Congress on Evolutionary Computation (CEC'14)*. 387–396. DOI: http://dx.doi.org/10.1109/CEC.2014.6900473

[61] Roberto E. Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Alexander Egyed, and Enrique Alba. 2016. *Evolutionary Computation for Software Product Line Testing: An Overview and Open Challenges*. Springer International, Cham, Switzerland, 59–87. DOI: http://dx.doi.org/10.1007/978-3-319-25964-2_4

[62] Roberto E. Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Alexander Egyed. 2015. A first systematic mapping study on combinatorial interaction testing for software product lines. In *Proceedings of the IEEE 8th International Conference on Software Testing, Verification, and Validation Workshops (ICSTW'15)*. IEEE, Los Alamitos, CA, 1–10.

[63] Roberto Erick Lopez-Herrejon, Javier Javier Ferrer, Francisco Chicano, Evelyn Nicole Haslinger, Alexander Egyed, and Enrique Alba. 2014. A parallel evolutionary algorithm for prioritized pairwise testing of software product lines. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO'14)*. ACM, New York, NY, 1255–1262. DOI: http://dx.doi.org/10.1145/2576768.2598305

[64] Roberto E. Lopez-Herrejon, Lukas Linsbauer, and Alexander Egyed. 2015. A systematic mapping study of search-based software engineering for software product lines. *Information & Software Technology* 61 (2015), 33–51.

[65] Víctor R. López-López, Leonardo Trujillo, and Pierrick Legrand. 2018. Novelty search for software improvement of a SLAM system. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO'18)*. ACM, New York, NY, 1598–1605. DOI: http://dx.doi.org/10.1145/3205651.3208237

[66] Chuan Luo, Binqi Sun, Bo Qiao, Junjie Chen, Hongyu Zhang, Jinkun Lin, Qingwei Lin, and Dongmei Zhang. 2021. LS-Sampling: An effective local search based sampling approach for achieving high t-wise coverage. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. ACM, New York, NY, 1081–1092.

[67] Chuan Luo, Qiyuan Zhao, Shaowei Cai, and Hongyu Zhangand Chunming Hu. 2022. SamplingCA: Effective and efficient sampling-based pairwise testing for highly configurable software systems. In *Proceedings of the ACM Joint*

European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22). ACM, New York, NY, 1–13. https://doi.org/10.1145/1122445.1122456

[68] Bogdan Marculescu, Robert Feldt, and Richard Torkar. 2016. Using exploration focused techniques to augment search-based software testing: An experimental evaluation. In *Proceedings of the 2016 IEEE International Conference on Software Testing, Verification, and Validation (ICST'16)*. 69–79. DOI: http://dx.doi.org/10.1109/ICST.2016.26

[69] Urtzi Markiegi, Aitor Arrieta, Goiuria Sagardui, and Leire Etxeberria. 2017. Search-based product line fault detection allocating test cases iteratively. In *Proceedings of the 21st International Systems and Software Product Line Conference—Volume A (SPLC'17)*. ACM, New York, NY, 123–132. DOI: http://dx.doi.org/10.1145/3106195.3106210

[70] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)*. 643–654.

[71] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. SPLOT: Software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. ACM, New York, NY, 761–762.

[72] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by Mapping Elites. *arXiv preprint arXiv:1504.04909* (2015).

[73] Jean-Baptiste Mouret and Glenn Maguire. 2020. Quality diversity for multi-task optimization. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO'20)*. ACM, New York, NY, 121–129. DOI: http://dx.doi.org/10.1145/3377930.3390203

[74] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. 2015. Innovation engines: Automated creativity and improved stochastic optimization via deep learning. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO'15)*. ACM, New York, NY, 959–966. DOI: http://dx.doi.org/10.1145/2739480.2754703

[75] Giuseppe Paolo, Alban Laflaquire, Alexandre Coninx, and Stephane Doncieux. 2020. Unsupervised learning and exploration of reachable outcome space. In *Proceedings of the 2020 IEEE International Conference on Robotics and Automation (ICRA'20)*. 2379–2385. DOI: http://dx.doi.org/10.1109/ICRA40945.2020.9196819

[76] Mike Papadakis, Christopher Henard, and Yves Le Traon. 2014. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *Proceedings of the 2014 IEEE 7th International Conference on Software Testing, Verification, and Validation*. 1–10. DOI: http://dx.doi.org/10.1109/ICST.2014.11

[77] José A. Parejo, Ana B. Sánchez, Sergio Segura, Antonio Ruiz-Cortés, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2016. Multi-objective test case prioritization in highly configurable systems: A case study. *Journal of Systems & Software* 122, C (2016), 287–310.

[78] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product sampling for product lines: The scalability challenge. In *Proceedings of the 23rd International Systems and Software Product Line Conference—Volume A (SPLC'19)*. ACM, New York, NY, 78–83.

[79] Thomas Pierrot, Guillaume Richard, Karim Beguir, and Antoine Cully. 2022. Multi-objective quality diversity optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'22)*. ACM, New York, NY, 139–147. DOI: http://dx.doi.org/10.1145/3512290.3528823

[80] Justin K. Pugh, Lisa B. Soros, and Kenneth O. Stanley. 2016. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI* 3 (2016), Article 40, 17 pages. DOI: http://dx.doi.org/10.3389/frobt.2016.00040

[81] Justin K. Pugh, L. B. Soros, Paul A. Szerlip, and Kenneth O. Stanley. 2015. Confronting the challenge of quality diversity. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO'15)*. ACM, New York, NY, 967–974. DOI: http://dx.doi.org/10.1145/2739480.2754664

[82] Vincenzo Riccio and Paolo Tonella. 2020. Model-based exploration of the frontier of behaviours for deep learning system testing. In *Proceedings of the 28th ACM Joint Meeting on the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. ACM, New York, NY, 876–888. DOI: http://dx.doi.org/10.1145/3368089.3409730

[83] José Raúl Romero, Aurora Ramírez, and Christopher L. Simons. 2019. Looking for novelty in SBSE problems. In *Proceedings of the Spanish Conference on Software Engineering and Databases (JISBD'19)*. 1–4.

[84] Ana B. Sánchez, Sergio Segura, José A. Parejo, and Antonio Ruiz-Cortés. 2017. Variability testing in the wild: The Drupal case study. *Software & Systems Modeling* 16, 1 (Feb. 2017), 173–194.

[85] Ana B. Sánchez, Sergio Segura, and Antonio Ruiz-Cortés. 2014. A comparison of test case prioritization criteria for software product lines. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST'14)*. IEEE, Los Alamitos, CA, 41–50. DOI: http://dx.doi.org/10.1109/ICST.2014.15

[86] Sergio Segura, José A. Galindo, David Benavides, José A. Parejo, and Antonio Ruiz-Cortés. 2012. BeTTy: Benchmarking and testing on the automated analysis of feature models. In *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'12)*. ACM, New York, NY, 63–71. DOI: http://dx.doi.org/10.1145/2110147.2110155

[87] Konstantinos Sfikas, Antonios Liapis, and Georgios N. Yannakakis. 2021. Monte Carlo elites: Quality-diversity selection as a multi-armed bandit problem. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'21)*. ACM, New York, NY, 180–188. DOI: http://dx.doi.org/10.1145/3449639.3459321

[88] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2011. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, 461–470. DOI: http://dx.doi.org/10.1145/1985793.1985856

[89] Andrei Strickler, Jackson A. Prado Lima, Silvia R. Vergilio, and Aurora T. R. Pozo. 2016. Deriving products for variability test of feature models with a hyper-heuristic approach. *Applied Soft Computing* 49 (2016), 1232–1242. DOI: https://doi.org/10.1016/j.asoc.2016.07.059

[90] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jensand Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85. DOI: https://doi.org/10.1016/j.scico.2012.06.002

[91] A. Vargha and H. D. Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[92] Omar M. Villanueva, Leonardo Trujillo, and Daniel E. Hernandez. 2020. Novelty search for automatic bug repair. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO'20)*. ACM, New York, NY, 1021–1028. DOI: http://dx.doi.org/10.1145/3377930.3389845

[93] Shuai Wang, Shaukat Ali, and Arnaud Gotlieb. 2013. Minimizing test suites in software product lines using weight-based genetic algorithms. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation (GECCO'13)*. ACM, New York, NY, 1493–1500. DOI: http://dx.doi.org/10.1145/2463372.2463545

[94] Shuai Wang, Shaukat Ali, and Arnaud Gotlieb. 2015. Cost-effective test suite minimization in product lines using search techniques. *Journal of Systems and Software* 103 (2015), 370–391. DOI: http://dx.doi.org/10.1016/j.jss.2014.08.024

[95] Shuai Wang, David Buchmann, Shaukat Ali, Arnaud Gotlieb, Dipesh Pradhan, and Marius Liaaen. 2014. Multi-objective test prioritization in software product line testing: An industrial case study. In *Proceedings of the 18th International Software Product Line Conference—Volume 1 (SPLC'14)*. ACM, New York, NY, 32–41. DOI: http://dx.doi.org/10.1145/2648511.2648515

[96] Yi Xiang, Han Huang, Miqing Li, Sizhe Li, and Xiaowei Yang. 2022. Looking for novelty in search-based software product line testing. *IEEE Transactions on Software Engineering* 48, 7 (2022), 2317–2338. DOI: http://dx.doi.org/10.1109/TSE.2021.3057853

[97] Yi Xiang, Xiaowei Yang, Han Huang, Zhengxin Huang, and Miqing Li. 2022. Sampling configurations from software product lines via probability-aware diversification and SAT solving. *Automated Software Engineering* 29, 2 (2022), 54.

[98] Zhihong Xu, Myra B. Cohen, Wayne Motycka, and Gregg Rothermel. 2013. Continuous test suite augmentation in software product lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*. ACM, New York, NY, 52–61. DOI: http://dx.doi.org/10.1145/2491627.2491650

[99] Qingfu Zhang and Hui Li. 2007. MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation* 11, 6 (2007), 712–731.

[100] Yulun Zhang, Matthew C. Fontaine, Amy K. Hoover, and Stefanos Nikolaidis. 2022. Deep surrogate assisted MAP-elites for automated hearthstone deckbuilding. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'22)*. ACM, New York, NY, 158–167. DOI: http://dx.doi.org/10.1145/3512290.3528718

[101] Tahereh Zohdinasab, Vincenzo Riccio, Alessio Gambi, and Paolo Tonella. 2021. DeepHyperion: Exploring the feature space of deep learning-based systems through illumination search. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*. ACM, New York, NY, 79–90. DOI: http://dx.doi.org/10.1145/3460319.3464811

[102] Tahereh Zohdinasab, Vincenzo Riccio, Alessio Gambi, and Paolo Tonella. 2023. Efficient and effective feature space exploration for testing deep learning systems. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), Article 49, 38 pages. DOI: http://dx.doi.org/10.1145/3544792

Received 20 November 2022; revised 25 July 2023; accepted 18 September 2023