CAmpactor: A Novel and Effective Local Search Algorithm for Optimizing Pairwise Covering Arrays

Qiyuan Zhao
Beihang University
Beijing, China
zqy1018@hotmail.com

Chuan Luo*
Beihang University
Beijing, China
chuanluo@buaa.edu.cn

Shaowei Cai
Institute of Software, Chinese Academy of Sciences
Beijing, China
caisw@ios.ac.cn

Wei Wu
Central South University and Xiangjiang Laboratory
Changsha, China
william.third.wu@gmail.com

Jinkun Lin
Seed Math Technology Limited
Beijing, China
linjk@seedmaas.com

Hongyu Zhang
Chongqing University
Chongqing, China
hyzhang@cqu.edu.cn

Chunming Hu
Beihang University
Beijing, China
hucm@buaa.edu.cn

ABSTRACT

The increasing demand for software customization has led to the development of highly configurable systems. Combinatorial interaction testing (CIT) is an effective method for testing these types of systems. The ultimate goal of CIT is to generate a test suite of acceptable size, called a t-wise covering array (CA), where t is the testing strength. Pairwise testing (*i.e.,* CIT with t=2) is recognized to be the most widely-used CIT technique and has strong fault detection capability. In pairwise testing, the most important problem is pairwise CA generation (PCAG), which is to generate a pairwise CA (PCA) of minimum size. However, existing state-of-the-art PCAG algorithms suffer from the severe scalability challenge; that is, they cannot tackle large-scale PCAG instances effectively, resulting in PCAs of large sizes. To alleviate this challenge, in this paper we propose *CAmpactor*, a novel and effective local search algorithm for compacting given PCAs into smaller sizes. Extensive experiments on a large number of real-world, public PCAG instances show that the sizes of *CAmpactor*'s generated PCAs are around 45% smaller than the sizes of PCAs constructed by existing state-of-the-art PCAG algorithms, indicating its superiority. Also, our evaluation confirms the generality of *CAmpactor*, since *CAmpactor* can reduce the sizes of PCAs generated by a variety of PCAG algorithms.

1 INTRODUCTION

The increasing demand for customized software and services has led to a growing interest in highly configurable systems [5, 31, 43, 48, 62, 63]. These systems allow users to customize their functionality by choosing different configurations of options. However, this flexibility also poses challenges for software testing, as the number of possible configurations can grow exponentially with the number of options. In practice, many real-world, highly configurable systems provide users with thousands of options to configure [5, 48, 50]. Given a highly configurable system with 1,000 options, where each option has 2 possible values, there could be 2^{1000} possible configurations, which is a huge number to test. Hence, in practice testing all possible configurations is usually infeasible.

Combinatorial interaction testing (CIT) is a practical and effective method for identifying faults in highly configurable systems [5]. CIT involves constructing a test suite (*i.e.,* a set of configurations) of a reasonable size and then using it to detect faults triggered by interactions of any t options, where t is the testing strength [48, 57, 79]. In particular, pairwise testing (*i.e.,* CIT with $t = 2$) is the most common CIT technique [50, 61], which is less costly than CIT with larger t values while still preserving high fault-detecting ability in practice [14, 26, 35, 36, 70, 80]. For a configurable system, a pairwise tuple represents the interaction of two options, and pairwise testing aims to build a pairwise covering array (PCA) of the given system, which is a test suite covering all pairwise tuples. It is crucial to use small-sized PCAs in real-world applications so that a large amount of testing effort could be saved.

In most real-world scenarios, there are also constraints (*e.g.,* mutual dependencies and exclusiveness) over configurable options. Test cases (*i.e.,* configurations) in a PCA must satisfy all constraints to guarantee the efficiency and accuracy of testing [64]. This requirement gives rise to the pairwise covering array generation (PCAG) problem, which aims to generate a minimum-sized PCA that satisfies all constraints, and remains a challenging problem in pairwise testing. Actually, highly configurable systems correspond to large-scale instances of the PCAG problem. Since the PCAG problem is recognized as a hard combinatorial optimization problem [39, 59], it is challenging to solve the PCAG problem for highly configurable systems, which urgently calls for practical solutions.

There are three major types of practical PCAG algorithms, *i.e.,* constraint-encoding algorithms (*e.g.,* [1, 4, 27, 78, 84]), greedy algorithms (*e.g.,* [8–10, 14, 33, 37–39, 72, 75, 77, 81]) and meta-heuristic algorithms (*e.g.,* [8, 15–17, 20, 22, 24, 30, 42, 43, 48, 55]). Although constraint-encoding algorithms can generate PCAs for small-scale PCAG instances, they cannot handle large-scale instances. Greedy algorithms can efficiently generate PCAs for medium-scale PCAG instances, but the generated PCAs are usually of large sizes. Thus, greedy algorithms are impractical in those application scenarios with limited testing budget. Compared to constraint-encoding and greedy algorithms, meta-heuristic algorithms can generate smaller-sized PCAs at the expense of long execution time. However, existing PCAG algorithms suffer from the severe scalability challenge

*Chuan Luo is the corresponding author of this work.

[50, 65, 76]. When handling highly configurable systems with thousands of options, a recent study demonstrates that these PCAG algorithms require a fairly long time to generate large-sized PCAs (*i.e.,* constructing large-sized test suites), which would greatly degrade the efficiency in testing highly configurable systems [50].

Local search is an effective meta-heuristic search paradigm for solving hard combinatorial optimization problems [12, 29, 45, 47, 49, 53, 68]. Many state-of-the-art meta-heuristic PCAG algorithms (*e.g.,* *AutoCCAG*[48], *FastCA*[42], *TCA*[43]) are based on local search [42]. These local search algorithms consist of two phases, *i.e.,* the initialization phase and the optimization phase. In the initialization phase, an efficient PCAG algorithm (*e.g.,* [14, 81]) is activated to construct an initial PCA in a short time. Then the optimization phase begins, where the initial PCA gets compacted (*i.e.,* its size gets reduced). Recently, Luo *et al.* proposed the state-of-the-art PCAG algorithm dubbed *SamplingCA* [50], which can generate PCAs for large-scale instances efficiently and therefore is a good candidate in the initialization phase. Although *SamplingCA* can generate PCAs of reasonable sizes, it does not explicitly incorporate any optimization technique for reducing the sizes of generated PCAs, leaving room for further optimization.

In this work, we propose *CAmpactor*, a novel and effective local search algorithm dedicated for compacting PCAs in the optimization phase. *CAmpactor* adopts a two-mode local search framework to optimize the sizes of input PCAs iteratively, and works between the exploitation mode and the exploration mode. In the exploitation mode, *CAmpactor* conducts optimization to search for a smaller-sized PCA, while in the exploration mode *CAmpactor* tends to explore promising search space. Given a PCA of size λ, *CAmpactor* attempts to find a PCA of size $\lambda - 1$. If the attempt succeeds (*i.e.,* a PCA of size $\lambda - 1$ is found), then *CAmpactor* continues to seek a PCA with a further smaller size (*e.g.,* $\lambda - 2$). By repeating this procedure, *CAmpactor* can generate a highly optimized PCA of smaller size. Furthermore, we propose two novel and powerful techniques (*i.e.,* assignment-level forbidden mechanism and forced patching technique) to further enhance the effectiveness of *CAmpactor* and alleviate the scalability challenge.

To evaluate *CAmpactor*, we conduct extensive experiments on a large number of publicly available PCAG instances encoded from real-world highly configurable systems. Our experiments present that, when using *SamplingCA* for initialization, *CAmpactor* can generate PCAs with around 45% smaller sizes than those from existing state-of-the-art PCAG algorithms (*i.e., SamplingCA*, *AutoCCAG*, *FastCA* and *TCA*), indicating the superiority of *CAmpactor*. Our results also confirm the effectiveness of core techniques used in *CAmpactor*. Further, the results present the generality of *CAmpactor*, since *CAmpactor* can significantly reduce the sizes of the PCAs generated by different PCAG algorithms. Our results show that *CAmpactor* can effectively mitigate the scalability challenge and significantly advance the state of the art in pairwise testing.

Our main contributions are summarized as follows.

- We propose *CAmpactor*, a novel and effective local search algorithm that can greatly reduce the size of input PCA.
- To enhance the performance of *CAmpactor*, we propose two novel and powerful techniques (*i.e.,* assignment-level forbidden mechanism and forced patching technique).
- We conduct extensive experiments to present the superiority of *CAmpactor* over existing state-of-the-art algorithms. Also, our results show the effectiveness of core techniques used in *CAmpactor*, as well as *CAmpactor*'s generality.

2 PRELIMINARIES

This section presents important definitions and notations.

2.1 Pairwise Covering Array Generation

A configurable system, *i.e.,* a *system under test* (SUT), can be configured by a finite set of *option*s, denoted as O. Each option $o_i \in O$ is related to its *value domain* Q_i, a finite set where o_i can take different values. As mentioned in Section 1, for real-world configurable systems there usually exists a set of *constraints* on the options [64], denoted as H, indicating the allowed combinations of values of the options in O. This work uses a pair in the form of (O, H) to represent an SUT.

For an SUT $S = (O, H)$, a *test case*, also known as a *configuration*, can be considered as a set of $|O|$ option-value pairs $T = \{(o_1, q_1), (o_2, q_2), \cdots, (o_{|O|}, q_{|O|})\}$, which assigns to each option $o_i \in O$ a value $q_i \in Q_i$. A test suite is a collection of multiple test cases. Similarly, a *pairwise tuple* is a set of exactly two option-value pairs $\tau = \{(o_i, q_i), (o_j, q_j)\}$, indicating that the options $o_i \in O, o_j \in O$ take the values $q_i \in Q_i, q_j \in Q_j$, respectively. A pairwise tuple τ is *covered* by a test case T if $\tau \subseteq T$, which means that the values of the two options in τ are the same with those ones in T. More generally, a pairwise tuple τ is *covered* by a test suite if τ is covered by at least one test case in that test suite. For a test case T and a pairwise tuple $\tau = \{(o_i, q_i), (o_j, q_j)\}$, we define the notation $T \lhd \tau$ to denote a new test case T', satisfying that a) T' covers τ and b) for any option o other than o_i and o_j, the values of option o in T and T' are the same. That is, $T \lhd \tau$ can be understood as a new test case that overrides T by τ.

Due to the existence of constraints, for an SUT $S = (O, H)$, a test case T is *valid* if T satisfies all constraints in H; otherwise (*i.e., T* violates at least one constraint in H), T is *invalid*. A pairwise tuple τ is *valid* if there exists at least one valid test case covering τ. In practice, invalid test cases would possibly incur incorrect testing results, so it is crucial to ensure that all test cases are valid.

Given an SUT $S = (O, H)$, a *pairwise covering array* (PCA) is a test suite (*i.e.,* a set of valid test cases), which covers all valid pairwise tuples. For a test suite A, if A is not a PCA, then A is called a *non-PCA test suite*. Also, a PCA or a test suite can be represented as a matrix, where each row is a valid test case, and each entry is called *cell* in this work. Particularly, for a test suite A, $U(A)$ denotes the collection of valid pairwise tuples that are not covered by A.

As a key problem in pairwise testing, the *pairwise covering array generation* (PCAG) problem is to find a minimum-sized PCA for the given SUT. It is recognized that the PCAG problem is a challenging combinatorial optimization problem [39, 59], which urgently calls for practical solutions.

Remark: Without loss of generality, following recent studies on highly configurable systems [5, 49, 50], this work concentrates on the scenario where each option takes Boolean values; that is, the value domains of all options are $\{0, 1\}$. Actually, it is well acknowledged that the general scenario, where the value domain of each

option is a set of multiple values, can be feasibly transformed into the binary scenario studied in this work [5, 49, 50]. Furthermore, the PCAG instances adopted in this work are of binary scenario, and all of them are transformed from the general scenario and encoded from practical, highly configurable systems. Hence, it is crucial to study the binary scenario in PCAG solving.

2.2 Boolean Formulae

Following recent work [2, 5, 6, 41, 49, 50, 56, 69], in this work highly configurable systems are modeled as Boolean formulae, whose necessary notations are introduced as follows.

Boolean variables are atoms of Boolean formulae. For a Boolean variable x, a *literal* of x is either x itself or its negation $\neg x$. A *Boolean formula* can be expressed in *conjunctive normal form* (CNF), *i.e.,* a conjunction of clauses, where a *clause* is a disjunction of literals. For a formula F in CNF, we use $V(F)$ and $C(F)$ to denote the set of all Boolean variables and the set of all clauses in F, respectively.

Given a Boolean variable x_i, its *value* v_i is either 0 or 1. The values of literals x_i and $\neg x_i$ are v_i and $1 - v_i$, respectively. An *assignment* of a formula F is a mapping $\alpha : V(F) \rightarrow \{0, 1\}$, which assigns a Boolean value to each variable. Given an assignment α, a clause c is *satisfied* if at least one literal in c evaluates to be 1 under α; otherwise, c is *unsatisfied*. Given an assignment α, if all clauses are satisfied under α, then α is a *solution* (*i.e.,* a satisfying assignment); otherwise, α is an unsatisfying assignment.

Given an SUT $S = (O, H)$, we can transform S into a formula F in CNF such that O and H correspond to $V(F)$ and $C(F)$, respectively. Through this way, a valid test case of S is indeed a solution (*i.e.,* a satisfying assignment) of F, and a pairwise tuple of S is a set of two literals of F (*e.g.,* the pairwise tuple $\{(o_1, 0), (o_2, 1)\}$ corresponds to the set of two literals $\{\neg x_1, x_2\}$).

Given an SUT S and its corresponding formula F, the PCAG problem aims to find a set of F's solutions such that all valid pairwise tuples of S are covered. In fact, seeking one solution for a formula in CNF is known as the influential, propositional satisfiability (SAT) problem [7], which is a challenging problem in theory. Hence, to solve the PCAG problem effectively, a practical SAT solver is indispensable.

A recently proposed SAT algorithm dubbed *ContextSAT* [50] shows its high performance in effectively generating valid test cases for highly configurable systems. *ContextSAT* [50] is developed based on *MiniSAT* [19], a well-known and effective SAT solver. The inputs of *ContextSAT* are as follows: a) F: a Boolean formula in CNF; b) L: a set of literals; c) α: an assignment of F. As described in *ContextSAT*'s literature [50], apart from these inputs, running *ContextSAT* needs to determine a variable order of $V(F)$, which guides the direction of the backtracking process. In this work, the variable order of *ContextSAT* is the one determined by *MiniSAT* [19]. In this setting, *ContextSAT* targets to seek a solution of F, which is similar to α (*i.e.,* many variables take the same values in *ContextSAT*'s solution and α) and guarantees that all literals in L evaluate to be 1. We use *ContextSAT*(F, L, α) to denote *ContextSAT*'s output solution. Particularly, given a valid pairwise tuple τ, *ContextSAT*(F, τ, α) denotes a satisfying assignment of F that covers τ and is similar to α.

Algorithm 1: Optimization Phase of Local Search for PCAG

Input: F: Boolean formula in CNF;
A: the initial PCA of F;
Output: A^*: the optimized PCA of F;

```
1 A* ← A;
2 while no termination criterion is met do
3     if A is a PCA of F then
4         A* ← A;
5         Remove an assignment from A;
6         continue ;
7     Perform one operation on A;
8 return A*;
```

3 LOCAL SEARCH PCAG ALGORITHMS

In this section, we describe the general framework of local search PCAG algorithms, and discuss their limitations.

3.1 General Framework of Local Search

As stated in Section 1, compared to other types of PCAG algorithms, meta-heuristic PCAG algorithms are more effective in minimizing the size of PCA. Also, state-of-the-art meta-heuristic PCAG algorithms (including *AutoCCAG* [48], *FastCA* [42] and *TCA* [43]) are based on the local search paradigm [42], and local search algorithms have achieved effectiveness in PCAG solving. Apart from the PCAG problem, local search has exhibited great success in solving various hard combinatorial optimization problems [12, 29, 45, 47, 49, 53, 68]. Therefore, our *CAmpactor* algorithm is also based on local search.

State-of-the-art local search PCAG algorithms (*e.g., AutoCCAG*, *FastCA* and *TCA*) consist of two phases, *i.e.,* initialization phase and optimization phase. In the initialization phase, an efficient PCAG algorithm is activated (*e.g.,* [14, 81]) to construct an initial PCA. Then, local search switches to the optimization phase that takes the initial PCA as input and optimizes the input PCA (*i.e.,* reduces the size of PCA).

The general framework of the optimization phase of local search PCAG algorithms is outlined in Algorithm 1. The output of this phase, denoted by A^*, is the smallest-sized PCA found during the optimization phase, and is updated with the initial PCA A in the beginning (Line 1 in Algorithm 1). In this phase, local search performs search steps iteratively until the termination criterion is met (Line 2 in Algorithm 1). In each iteration (*i.e.,* search step), local search first checks whether A is a PCA (Line 3 in Algorithm 1). If so, local search updates A^* accordingly, and reduces the size of A by 1 through removing an assignment (*i.e.,* a test case) from A, which possibly makes A no longer be a PCA (Line 4-6 in Algorithm 1). Otherwise (*i.e., A* is not a PCA), local search tries to make A become a PCA by performing one operation on A (Line 7 in Algorithm 1).[1] When the termination criterion is met, local search stops and outputs A^* (Line 8 in Algorithm 1).

The main idea of the optimization phase is summarized as follows: local search aims to seek a PCA of a specific size λ; once a λ-sized PCA is found, then local search tries to find a PCA of size

[1] In this work, performing one operation on A means modifying an assignment $\alpha \in A$ via changing the values of cells in α.

$\lambda - 1$. Hence, the size of A^* is decreasing during the search, and A^* is essentially the minimum-sized PCA that local search can find.

3.2 Limitations of Existing Local Search

As introduced in Section 1, existing PCAG algorithms, including local search ones, suffer from the severe scalability challenge. In particular, when dealing with highly configurable systems with thousands of options, a recent empirical study presents that existing state-of-the-art local search algorithms fail to generate PCAs of small sizes [50]. Hence, before introducing *CAmpactor*, we identify two issues that make existing local search algorithms suffer from the scalability challenge, and discuss their possible solutions.

3.2.1 Cycling Issue. In the context of local search for combinatorial optimization problems, the cycling issue [58], referring to the circumstance where local search stagnates in a local part of search space, is a major weakness of local search, and severely degrades the performance of local search [11].

To handle the cycling issue, it is advisable to equip local search with effective forbidden mechanisms. Actually, state-of-the-art local search PCAG algorithms [42, 43, 48] adopt the well-known forbidden mechanism, *i.e.,* cell-level tabu mechanism [43], to handle the cycling issue. In particular, the cell-level tabu mechanism prohibits such operations aiming to modify cells whose values have been changed recently [43]. When solving large-scale PCAG instances, since there are a large number of candidate operations, such cell-level tabu mechanism can only prohibit few of them, so its forbidden strength is relatively weak. However, it is known that, for a forbidden mechanism, weak forbidden strength (*i.e.,* only few operations are impermissible) would still make local search suffer from the cycling issue [44, 46]. Unfortunately, in the context of PCAG solving, little attention has been paid on enhancing forbidden strategies. Thus, an effective solution to the cycling issue is to strengthen forbidden mechanisms.

3.2.2 Hindering Issue. Given a non-PCA test suite A, in each search step existing state-of-the-art local search PCAG algorithms (*e.g., AutoCCAG* [48], *FastCA*[42] and *TCA*[43]) perform a minor operation on A, where at most two cells in an assignment $\alpha \in A$ get their values changed. Particularly, in each search step, existing local search PCAG algorithms select one uncovered pairwise tuple $\tau \in U(A)$ randomly, and attempt to cover τ by performing one minor operation on A while preserving the validity of test cases in A. However, recent studies present that practical SUTs have many constraints over options (*e.g.,* the well-known FreeBSD operating system has 62,183 constraints) [5, 49, 50]. Due to the existence of complex constraints, it is frequently infeasible to cover τ via performing only one minor operation on A while keeping that all test cases in A are valid. This hindering issue greatly prevents local search from being effective.

To alleviate the hindering issue, rather than performing a minor operation in a search step as existing local search PCAG algorithms do, it is desirable to perform a major operation, where more than two cells in an assignment $\alpha \in A$ get their values changed, to cover τ in a search step while satisfying all constraints.

Algorithm 2: The *CAmpactor* Algorithm

Input: F: Boolean formula in CNF;
A: the initial PCA of F;
Output: A^*: the optimized PCA of F;

$A^* \leftarrow A$;
$budget \leftarrow \gamma$;
while $budget > 0$ **do**
  if *A is a PCA of F* **then**
    $A^* \leftarrow A$;
    Remove the assignment with the smallest loss from A;
    $budget \leftarrow \gamma$;
    continue;
  $budget \leftarrow budget - 1$;
  $\tau \leftarrow$ a random uncovered pairwise tuple in $U(A)$;
  $R = \{(\alpha, \alpha \lhd \tau) \mid \alpha \in A, age(\alpha) > \delta, \alpha \lhd \tau \text{ is valid}\}$;
  if $R \neq \emptyset$ **then**
    $(\alpha^*, \alpha^* \lhd \tau) \leftarrow$ operation with the largest profit in R;
    Perform operation $(\alpha^*, \alpha^* \lhd \tau)$ on A;
  else if *with probability* ψ **then**
    $M = \{(\alpha, ContextSAT(F, \tau, \alpha)) \mid \alpha \in A, age(\alpha) > \delta\}$;
    $(\alpha^*, ContextSAT(F, \tau, \alpha^*)) \leftarrow$ operation with the largest profit in M;
    Perform operation $(\alpha^*, ContextSAT(F, \tau, \alpha^*))$ on A;
  else
    foreach *cell e of A with a random order* **do**
      $\alpha \leftarrow$ the assignment where cell e locates;
      $\alpha' \leftarrow \alpha$ with cell e modified;
      if α' *is a satisfying assignment* **then**
        Perform operation (α, α') on A;
        break;
return A^*;

4 OUR PROPOSED *CAMPACTOR* ALGORITHM

In this section, we propose *CAmpactor*, a novel and effective local search algorithm for reducing the size of given PCA.

4.1 Overall Design of *CAmpactor*

Based on the general framework of local search PCAG algorithms outlined in Algorithm 1, we propose *CAmpactor* that is dedicated to reducing the size of the given PCA. The overall design of *CAmpactor* is presented in Algorithm 2. *CAmpactor* requires two inputs: a) F, a Boolean formula in CNF that is transformed from an SUT and b) A, the initial PCA of F generated in the initialization phase. The output of *CAmpactor* is A^*, an optimized PCA of F.

According to the general framework in Algorithm 1, there are three essential components: a) the termination criterion, b) the strategy to select the assignment to be removed from A when A is a PCA, and c) the mechanism to determine what operation to be performed on A. We specify and introduce these components in *CAmpactor* as follows.

4.1.1 Termination Criterion. For *CAmpactor*, we design a budget-aware termination criterion, which works as follows. In the beginning, the search budget is initialized as a large-valued positive

integer γ (Line 2 in Algorithm 2). Once a search step (*i.e.,* an iteration) has been conducted, then the search budget decreases by 1 (Line 9 in Algorithm 2). Particularly, whenever A is a PCA, then the search budget is reset to γ (Line 7 in Algorithm 2). During the local search process, once the search budget reaches 0, then the entire search process of *CAmpactor* terminates (Line 3 in Algorithm 2).

According to our budget-aware termination criterion described above, *CAmpactor* terminates if and only if it does not find a smaller-sized PCA in consecutive γ search steps. Actually, γ is an integer-valued hyper-parameter in *CAmpactor*, and adjusting γ can balance the efficiency and effectiveness of *CAmpactor*. With a large-valued γ, *CAmpactor* takes longer running time but outputs a smaller-sized PCA. Conversely, with a small-valued γ, *CAmpactor* terminates quickly, while the generated PCA is possibly of larger size. The effect of γ will be investigated empirically in Section 6.3.

4.1.2 Removal Strategy. Before describing the removal strategy, we first define the *loss* of assignment, a key concept in *CAmpactor*. For a PCA A, the loss of an assignment $\alpha \in A$ is the number of covered pairwise tuples that would become uncovered, *i.e.,* the increment of $|U(A)|$, if α is removed from A.

Intuitively, in the search process, if there exist more uncovered tuples, then more operations need to be performed on A to make A become a PCA. Hence, *CAmpactor* adopts a greedy removal strategy, which works as follows: once A becomes a PCA, *CAmpactor* selects the assignment $\alpha \in A$ with the smallest loss to be removed from A (Line 6 in Algorithm 2).

4.1.3 Operation Determination Method. The operation determination method is the most important component in *CAmpactor*, since it directly guides *CAmpactor*'s search direction and thus its effectiveness greatly affects the size of output PCA. In fact, two-mode local search algorithms, which work between the exploitation mode and the exploration mode, have achieved great success in solving various combinatorial optimization problems [29, 40, 44, 45, 58]. Therefore, *CAmpactor* follows the two-mode design. Generally, in the exploitation mode local search tends to optimize the objective, while in the exploration mode local search targets to diversify the search direction. More precisely, in the exploitation mode *CAmpactor* prefers to conduct those operations that can reduce the number of uncovered pairwise tuples, *i.e.,* trying to make A become a PCA (Lines 10–18 in Algorithm 2). In the exploration mode *CAmpactor* aims to perform such operations that can better explore the promising search space (Lines 20–25 in Algorithm 2). In this way, *CAmpactor* can achieve a good balance between exploitation and exploration. The technical details of both exploitation mode and exploration mode are described in the following subsections.

4.2 Exploitation Mode

As aforementioned, in the exploitation mode *CAmpactor* conducts operations to reduce the number of uncovered pairwise tuples (*i.e.,* decreasing $|U(A)|$). It is advisable to design the exploitation mode as follows: first, an uncovered pairwise tuple $\tau \in U(A)$ is randomly selected, and then an operation, which modifies an assignment $\alpha \in A$ to another valid assignment α' that covers τ, is performed. Thus, in this work an operation can be expressed by a combination of two assignments (α, α'), which means modifying α to α' in A. Based on this design, at least one uncovered tuple (*i.e.,* τ) is ensured to be covered, so it is possible that $|U(A)|$ (*i.e.,* the number of uncovered pairwise tuples) could be reduced.

Since there usually exist more than one candidate operation covering τ, it is necessary to utilize a metric to assess the benefit of an operation, such that the best candidate operation can be selected. Thus, we adopt a metric called *profit* to assess an operation's benefit. Given an operation (α, α'), the profit of the operation, denoted by $profit(\alpha, \alpha')$, is the decrement in the number of uncovered pairwise tuples, if the operation is performed (*i.e.,* α is modified to α' in A); particularly, $profit(\alpha, \alpha')$ is calculated as the number of uncovered pairwise tuples becoming covered minus the number of covered tuples becoming uncovered, if operation (α, α') is taken. Clearly, performing an operation with larger profit brings more benefit.

4.2.1 Assignment-level Forbidden Mechanism. It is recognized that the cycling issue greatly degrades the performance of local search, and adopting forbidden strategies can reduce the negative consequence. In fact, existing state-of-the-art PCAG algorithms [42, 43, 48] adopt a cell-level tabu mechanism [43], which prevents modifying several cells that have been changed recently. As discussed in Section 3.2.1, due to its weak forbidden strength, such cell-level tabu mechanism is ineffective in handling the cycling issue. For the empirical evidence, readers can refer to Section 6.2.

For *CAmpactor* we propose a novel, assignment-level forbidden mechanism with strong forbidden strength. Before introducing our new forbidden mechanism, we define an important notion, *i.e.,* the age of assignment. Given an assignment $\alpha \in A$, the age of α, denoted by $age(\alpha)$, is the number of search steps that have been conducted since the last modification of α. Particularly, our assignment-level forbidden mechanism is designed to strictly prohibit such an operation which aims to modify an assignment α that has been changed in the last δ search steps (*i.e.,* $age(\alpha) \leq \delta$), where δ is a positive integer. Following our new forbidden mechanism, once the uncovered pairwise tuple $\tau \in U(A)$ is selected, *CAmpactor* constructs a candidate set R of operations as $R = \{(\alpha, \alpha \lhd \tau) \mid \alpha \in A, age(\alpha) > \delta, \alpha \lhd \tau \text{ is valid}\}$, recalling that the operator $\lhd$ is defined in Section 2.1, and $\alpha \lhd \tau$ can be understood as an assignment that overrides α by τ. Then *CAmpactor* adopts a greedy operation determination method: after the candidate set R is constructed, *CAmpactor* selects and performs the operation $(\alpha^*, \alpha^* \lhd \tau)$ with the largest profit in R.

When solving a large-scale PCAG instance (which indicates that an assignment has a large number of cells), compared to the existing cell-level tabu mechanism, our assignment-level forbidden mechanism can prohibit more operations, and thus has stronger forbidden strength. In addition, δ is a hyper-parameter in *CAmpactor* that controls the forbidden strength, and its effect will be analyzed in Section 6.3.

4.2.2 Forced Patching Technique. In fact, the operation selected by our forbidden mechanism is a minor operation where an assignment α^* is modified to $\alpha^* \lhd \tau$. However, as described in Section 3.2.2, due to the existence of complex constraints, the hindering issue frequently causes that no feasible, minor operation can be performed to cover the selected, uncovered pairwise tuple τ, which

is a crucial problem that prevents existing local search PCAG algorithms from being effective. Thus, the candidate set R can be empty, as a frequent scenario.

Once R is empty, a natural solution is to abandon covering τ, and instead to take a minor operation that does not violate any constraint, so as to continue the search process. That is, the natural solution directly ignores such crucial problem in the current search step. However, since τ cannot be covered by any minor operation, the same problem would therefore occur in subsequent search steps.

Hence, an advisable solution is to perform a major operation to make A cover τ by force. Here we need to address the core technical challenge, *i.e.,* how to find appropriate operations that cover τ and meanwhile make all constraints satisfied.

To address this challenge, we propose an effective forced patching technique, which leverages the power of an SAT solver, since an SAT solver can find a satisfying assignment that covers a given pairwise tuple for a given formula. As described in Section 2.2, *ContextSAT* [50] is an effective SAT solver; also, given a formula F in CNF, a valid pairwise tuple τ and an assignment α of F as inputs, *ContextSAT* can return a satisfying assignment of F that covers τ and is similar to α, denoted by $ContextSAT(F, \tau, \alpha)$. Since we aim to find such operations that modify an assignment $\alpha \in A$ to cover τ and meanwhile to satisfy all constraints appearing in F, an advisable solution is as follows: for each assignment $\alpha \in A$ with $age(\alpha) > \delta$, a candidate operation $(\alpha, ContextSAT(F, \tau, \alpha))$ can be constructed. Actually, apart from making A cover τ, another important objective is to make A cover more pairwise tuples. Thanks to the property that both assignments α and $ContextSAT(F, \tau, \alpha)$ are similar, it is clear that many pairwise tuples exclusively covered by α can be preserved if the operation $(\alpha, ContextSAT(F, \tau, \alpha))$ is performed. In a nutshell, we can construct a candidate set M of major operations, *i.e.,* $M = \{(\alpha, ContextSAT(F, \tau, \alpha)) \mid \alpha \in A, age(\alpha) > \delta\}$.

Once the candidate set M of major operations is constructed, from the candidate set M, *CAmpactor* selects and performs the best operation $(\alpha^*, ContextSAT(F, \tau, \alpha^*))$ with the largest profit. It is clear that activating the forced patching technique would call the *ContextSAT* solver multiple times. Since calling *ContextSAT* requires a certain amount of computation time, in order to keep the efficiency of *CAmpactor*, we activate the forced patching technique with a fixed probability ψ. That is, with a fixed probability ψ, *CAmpactor* calls the forced patching technique; otherwise (*i.e.,* with a probability $1 - \psi$), *CAmpactor* works in the exploration mode, which will be introduced in the next subsection. Here ψ is a real-valued hyper-parameter of *CAmpactor*, and its value domain ranges from 0 to 1. Since ψ plays a key role in balancing the effectiveness and efficiency of *CAmpactor*, its effect will be studied in Section 6.3.

4.3 Exploration Mode

It is clear that the exploitation mode of *CAmpactor* aims to modify the test suite A in a greedy manner. However, existing studies present that only adopting greedy strategies could make local search get stuck in a small part of search space [3, 28, 54]. Thus, it is desirable to incorporate exploration mode into local search.

The task of *CAmpactor*'s exploration mode is to explore the promising part of search space. Also, in the context of local search for solving combinatorial optimization problems, applying randomized strategies in the exploration mode can help local search better explore promising search space [29, 40, 44, 45, 58]. Based on this design, in the exploration mode *CAmpactor* works as follows. *CAmpactor* tries to traverse all cells of A with a random order. For each cell e of A, *CAmpactor* selects the assignment α where cell e locates, and tries to modify cell e in α, resulting in a new assignment α'. Once α' is a satisfying assignment, *CAmpactor* performs the minor operation (α, α') on A, and then terminates its exploration mode. Through this way, *CAmpactor* is able to generally perform a random, minor operation.

4.4 Discussions

In this subsection, we discuss how *CAmpactor* deals with the cycling issue and the hindering issue.

***CAmpactor*'s solution to the cycling issue:** In Section 4.2.1, *CAmpactor* utilizes a novel, assignment-level forbidden mechanism, which has stronger forbidden strength than existing cell-level tabu mechanism. As discussed in Section 3.2.1, forbidden mechanisms with strong forbidden strength can tackle the cycling issue. Therefore, compared to existing local search algorithms that adopt the cell-level tabu mechanism, *CAmpactor* can better alleviate the cycling issue through our assignment-level forbidden mechanism.

***CAmpactor*'s solution to the hindering issue:** From Section 3.2.2, performing major operations can help mitigate the hindering issue. Compared to existing local search PCAG algorithms that generally perform minor operations, *CAmpactor* can better handle the hindering issue, through complementing the minor operations with appropriate major operations, which are determined by our forced patching technique (as introduced in Section 4.2.2).

As discussed in Section 3.2, both cycling issue and hindering issue cause existing local search algorithms suffer from the scalability challenge. Since *CAmpactor* can tackle both issues more effectively than existing local search algorithms, *CAmpactor* is able to better alleviate the scalability challenge, which is confirmed by our evaluation presented in Section 6.

5 EXPERIMENTAL PRELIMINARIES

This section describes the experimental preliminaries of this work.

5.1 Public PCAG Instances

In our experiments, we adopt a collection of 124 public PCAG instances. Each PCAG instance is derived from a practical, highly configurable system, and is encoded as a Boolean formula in CNF. These PCAG instances are originally presented by Baranov *et al.* [5], and have been broadly evaluated in recent studies on testing highly configurable systems [5, 32, 41, 49, 50, 60, 65, 66]. For all PCAG instances, the numbers of options range from 94 to 11,254, and the numbers of constraints vary from 190 to 62,183. To help readers better reproduce our experiments, all PCAG instances tested in our experiments and the information of each instance (*i.e.,* the numbers of options and constraints) are publicly available in our public repository.[2]

[2]https://github.com/chuanluocs/CAmpactor

5.2 State-of-the-art PCAG Algorithms

In this work, *CAmpactor* is compared against four state-of-the-art PCAG algorithms, *i.e., SamplingCA* [50], *AutoCCAG* [48], *FastCA* [42] and *TCA* [43], which are described as follows.

SamplingCA [50] is a recently proposed, sampling based PCAG algorithm, and represents the current state of the art in solving the PCAG problem. The experiments reported in the literature [50] show that *SamplingCA* performs much better than all other PCAG algorithms (including *AutoCCAG*, *FastCA* and *TCA*) when dealing with many highly configurable systems. The implementation of *SamplingCA* is publicly available.[3]

AutoCCAG [48] is an advanced approach based on automated algorithm optimization. As summarized in the literature [48], the performance of *AutoCCAG* is much better than that of other well-known methods (including *CASA* [20, 21], *TCA* [43] and *CHiP* [57]) on many real-world instances. The source code of *AutoCCAG* is obtained from its authors [48].

FastCA [42] is an effective meta-heuristic algorithm. As reported in the literature [42], *FastCA* exhibits better performance than a variety of effective algorithms (including *TCA*, *CASA*, *ACTS* [81] and *HHSA* [30]) on extensive application instances. The source code of *FastCA* can be obtained online.[4]

TCA [43] is a high-performance meta-heuristic algorithm. As demonstrated in the literature [43], *TCA* greatly reduces the size of PCAs compared to a number of influential algorithms (including *CASA*, *ACTS* and *Cascade* [84]) on plenty of practical instances. The implementation of *TCA* is publicly available online.[5]

Also, we conduct experiments to measure the performance of four other influential PCAG algorithms, *i.e., HHSA* [30], *CASA* [20, 21], *ACTS* [81] and *CTLog* [1]. Our evaluation results show that *HHSA*, *CASA*, *ACTS* and *CTLog* fail to construct PCAs for the majority of the testing instances. To save space, we do not report their results in this paper. For the full results of *HHSA*, *CASA*, *ACTS* and *CTLog*, readers can refer to our public repository.[2]

5.3 Research Questions

This work is devoted to advancing the state of the art in solving the PCAG problem. For solving PCAG, the ultimate target is to reduce the size of generated PCA. Hence, our experiments focus on minimizing the size of generated PCA. We aim to answer the following research questions (RQs).

RQ1. Is *CAmpactor* able to generate smaller-sized PCAs than its state-of-the-art competitors?

In this RQ, we compare *CAmpactor* against four state-of-the-art PCAG algorithms, *i.e., SamplingCA*, *AutoCCAG*, *FastCA* and *TCA* on a variety of public PCAG instances.

RQ2. Does each core technique of *CAmpactor* contribute to the performance improvement of *CAmpactor*?

In this RQ, we conduct extensive experiments to analyze the effectiveness of all core algorithmic techniques of *CAmpactor*, *i.e.,* assignment-level forbidden mechanism (in Section 4.2.1) and forced patching technique (in Section 4.2.2).

RQ3. How does each hyper-parameter of *CAmpactor* impact the effectiveness of *CAmpactor*?

In this RQ, we conduct empirical evaluation to explore how the settings of all *CAmpactor*'s hyper-parameters, *i.e.,* ψ, δ and γ, impact the effectiveness of *CAmpactor*.

RQ4. Can *CAmpactor* further reduce the sizes of PCAs that are generated by different PCAG algorithms?

In this RQ, we empirically study whether *CAmpactor* can optimize the PCAs constructed by different PCAG algorithms.

5.4 Experimental Design

Hardware environment: In this work, all experiments are performed on a computing workstation with 2.60GHz Intel Xeon Platinum 8171M CPU and 256GB memory, running the operating system of Ubuntu 18.04.4 LTS.

Setting of initialization algorithm: According to Section 4, *CAmpactor* aims to reduce the sizes of PCAs. Hence, a PCAG algorithm is first used to initialize a PCA as *CAmpactor*'s input; then *CAmpactor* is activated to optimize such initialized PCA, and the output of *CAmpactor* is regarded as the final, optimized PCA to be reported. Since a recent study [50] demonstrates that *SamplingCA* is able to generate smaller-sized PCAs than *AutoCCAG*, *FastCA* and *TCA*, in this work *CAmpactor* adopts *SamplingCA* as its initialization algorithm. For simplicity, in our experiments we use *CAmpactor* to directly represent *CAmpactor*'s instantiation that generates its initial PCA through *SamplingCA*. In fact, *CAmpactor* treats the initialization algorithm as a black box, so *CAmpactor* is able to integrate any other PCAG algorithms as its initialization algorithm. To study the generality of *CAmpactor* (*i.e.,* whether *CAmpactor* can further compact the PCAs output by different PCAG algorithms), the performance of *CAmpactor* adopting *AutoCCAG*, *FastCA* and *TCA* as its initialization algorithms will be shown in Section 6.4.

Since *CAmpactor* employs *SamplingCA* as its initialization algorithm, in order to make our comparison fair, *AutoCCAG*, *FastCA* and *TCA* all replace their own initialization algorithms with *SamplingCA* in our experiments. We have conducted the evaluation to compare the versions of *AutoCCAG*, *FastCA* and *TCA*, which utilize *SamplingCA* as their initialization algorithms, against the original versions of *AutoCCAG*, *FastCA* and *TCA*, which use their own, original initialization algorithms. Actually, according to our evaluation, through adopting *SamplingCA* as the initialization algorithm, *AutoCCAG*, *FastCA* and *TCA* achieve considerable performance improvement. To save space, we do not report these results in this paper. For the full comparative results, readers can refer to our public repository.[2] For simplicity, in our experiments we directly use *AutoCCAG*, *FastCA* and *TCA* to denote their versions that construct initial PCAs through *SamplingCA*.

Experimental setup: For *CAmpactor*, its hyper-parameters γ, ψ and δ are set to 10,000, 0.1 and 10, respectively, and the effect of *CAmpactor*'s each hyper-parameter will be analyzed in Section 6.3. The implementation of *CAmpactor* is available in our public repository.[2] In our experiments, *SamplingCA*, *AutoCCAG*, *FastCA* and *TCA* are evaluated using the hyper-parameter settings recommended by their authors [42, 43, 48, 50].

[3] https://github.com/chuanluocs/SamplingCA
[4] https://github.com/jkunlin/fastca
[5] https://github.com/jkunlin/TCA

Table 1: Results of *CAmpactor*, *CAmpactor-Short*, *SamplingCA*, *AutoCCAG*, *FastCA* and *TCA* on 20 selected PCAG instances.

Benchmarks	*CAmpactor*		*CAmpactor-Short*		*SamplingCA*		*AutoCCAG*		*FastCA*		*TCA*	
	min.(avg.)	time (s)	min.(avg.)	time (s)	min.(avg.)	time (s)	min.(avg.)	time (s)	min.(avg.)	time (s)	min.(avg.)	time (s)
busybox_1_28_0	**24 (24.7)**	64.3	42 (44.5)	14.1	57 (58.8)	11.5	45 (47.4)	637.6	42 (46.7)	19.2	53 (55.6)	14.1
calm16_ceb	**43 (45.2)**	215.7	77 (81.1)	33.0	100 (103.0)	26.7	81 (83.9)	635.7	81 (84.5)	383.7	93 (97.0)	33.0
cq7750	**44 (46.6)**	242.0	81 (84.8)	37.7	103 (105.3)	31.4	84 (87.9)	377.0	85 (88.1)	341.2	92 (99.2)	37.7
dreamcast	**50 (53.2)**	273.9	87 (94.1)	40.1	109 (113.4)	33.7	89 (95.2)	54.6	89 (95.6)	391.5	102 (107.7)	40.1
eb40a	**45 (46.4)**	226.7	77 (80.8)	37.0	99 (103.6)	29.9	81 (85.1)	498.8	81 (85.0)	630.4	94 (97.2)	37.0
ecos-icse11	**47 (48.7)**	242.8	81 (84.4)	39.6	107 (108.4)	31.1	88 (91.9)	589.9	89 (92.4)	352.3	99 (102.0)	39.6
freebsd-icse11	**53 (56.2)**	354.4	73 (77.9)	74.5	118 (121.7)	43.8	78 (82.2)	2639.1	81 (83.4)	2076.6	101 (108.4)	74.5
integrator_arm9	**58 (60.1)**	324.9	103 (105.3)	43.3	118 (122.1)	37.1	101 (105.0)	383.1	101 (105.1)	886.7	115 (117.1)	43.3
linux	**51 (53.9)**	279.2	91 (98.1)	38.2	111 (115.3)	32.9	94 (97.6)	257.0	92 (97.7)	163.2	106 (110.0)	38.2
mb93091	**47 (48.4)**	307.0	83 (86.5)	38.3	103 (106.3)	31.1	85 (89.3)	325.9	84 (89.2)	493.5	97 (101.5)	38.3
mpc50	**43 (45.0)**	253.3	77 (80.3)	34.8	96 (100.9)	28.3	80 (83.3)	99.5	79 (83.1)	383.3	86 (95.2)	34.8
olpce2294	**48 (49.7)**	314.2	83 (85.6)	43.8	106 (109.8)	33.4	91 (93.4)	414.1	91 (92.9)	509.6	102 (103.9)	43.8
pc_i82544	**48 (49.8)**	314.2	82 (84.7)	44.3	107 (109.9)	32.5	89 (93.9)	50.4	91 (93.7)	342.7	98 (104.2)	44.3
pc_usb_d12	**45 (47.1)**	278.8	78 (81.1)	41.9	102 (105.8)	32.4	82 (89.2)	46.7	82 (88.8)	545.7	95 (100.2)	41.9
refidt334	**52 (54.6)**	252.8	84 (89.5)	43.3	110 (113.5)	33.8	89 (94.5)	677.0	88 (94.0)	478.5	101 (106.6)	43.3
sam7ex256	**47 (50.3)**	293.2	86 (88.2)	45.9	105 (109.6)	36.4	91 (93.5)	432.7	91 (93.5)	177.1	102 (104.5)	45.9
sleb	**43 (44.5)**	229.7	71 (74.7)	34.1	98 (99.3)	26.2	78 (81.4)	322.7	78 (81.4)	457.7	90 (93.5)	34.1
uClinux-config	**36 (37.6)**	3495.2	52 (52.9)	1856.1	64 (66.5)	1493.0	64 (65.4)	1856.5	64 (65.4)	1613.3	64 (65.4)	1856.1
vrc4375	**48 (49.8)**	257.8	85 (88.8)	39.5	105 (109.3)	32.4	89 (91.9)	98.8	89 (91.4)	364.7	99 (104.0)	39.5
XSEngine	**48 (49.7)**	277.5	81 (83.5)	41.1	104 (107.7)	32.1	86 (89.5)	236.2	87 (89.4)	118.6	98 (101.1)	41.1

Table 2: Average size and average running time of *CAmpactor*, *CAmpactor-Short*, *SamplingCA*, *AutoCCAG*, *FastCA* and *TCA* over all PCAG instances.

	CAmpactor	*Short**	*SamplingCA*	*AutoCCAG*	*FastCA*	*TCA*
avg. size	**47.4**	82.7	104.0	86.6	86.7	98.1
avg. time	284.6	52.7	42.1	377.9	357.2	52.7

*To save space, we use '*Short*' to denote '*CAmpactor-Short*'.

Table 3: Average size and average running time of *CAmpactor* and all its alternative versions over all instances.

	CAmpactor	*Alt-1*	*Alt-2*	*Alt-3*
avg. size	**47.4**	79.8	54.3	98.6
avg. time	284.6	67.3	278.7	46.1

Since *CAmpactor* and all its state-of-the-art competitors, *i.e.*, *SamplingCA*, *AutoCCAG*, *FastCA* and *TCA*, are randomized algorithms, we conduct 10 independent runs per instance for each algorithm. In our experiments, the cutoff time for each algorithm run is set to 3,600 CPU seconds, as suggested by a recent study on solving hard combinatorial optimization problems [83].

For each algorithm on solving each PCAG instance, we report the minimum size of the output PCAs among 10 runs, denoted by 'min', the average size of the output PCAs over 10 runs, denoted by 'avg', and the running time measured in CPU seconds averaged over 10 runs, denoted by 'time'. In our experiments, the running time of *CAmpactor*, *AutoCCAG*, *FastCA* and *TCA* include the computational time of their initialization algorithms. To study the overall performance of all algorithms, for each algorithm, we present the average size and the average running time over the whole PCAG instance collection. Because the ultimate objective of the PCAG problem is to minimize the size of generated PCA, in our experiments, for each PCAG instance or the whole PCAG instance collection, if an algorithm generates the smallest-sized PCA among all competing algorithms, its results of 'min' and 'avg' are highlighted using the **boldface** font.

Statistical significance and effect size calculation: Furthermore, as suggested by recent empirical studies [48–50], in our experiments, for each PCAG instance or the whole PCAG instance collection, we perform the Wilcoxon signed-rank test [18] to examine the statistical significance of any pairwise comparison between *CAmpactor* and each of its competitors, and we calculate the Vargha-Delaney effect size [73] for each pairwise comparison. For each PCAG instance or the whole PCAG instance collection, if a) all the p-values of Wilcoxon signed-rank tests at 95% confidence level are smaller than 0.05, and b) the Vargha-Delaney effect sizes for all pairwise comparisons (between *CAmpactor* and each of its competitors) are larger than 0.71 (implying large effect sizes) [48–50, 67, 73], we consider that the performance improvement of *CAmpactor* over all its competitors is both statistically significant and meaningful, and *CAmpactor*'s results are indicated using underline.

6 EXPERIMENTAL RESULTS

This section reports and analyzes the experimental results.

6.1 Comparisons with State of the Art (RQ1)

The comparative results of *CAmpactor* and existing state-of-the-art PCAG algorithms (including *SamplingCA*, *AutoCCAG*, *FastCA* and *TCA*) on 20 selected PCAG instances are reported in Table 1. For these 20 selected PCAG instances, 10 of them are identified to be representative in a recent study [5], and the other 10 instances

Table 4: Average size and average running time of *CAmpactor* with various hyper-parameter settings of ψ over all instances.

	$\psi = 0$	$\psi = 0.1$	$\psi = 0.2$	$\psi = 0.3$	$\psi = 0.4$	$\psi = 0.5$	$\psi = 0.6$	$\psi = 0.7$	$\psi = 0.8$	$\psi = 0.9$	$\psi = 1.0$
avg. size	98.6	47.4	46.6	46.1	45.9	45.7	45.7	45.5	45.5	**45.4**	**45.4**
avg. time	46.1	284.6	362.2	440.8	503.6	569.0	610.8	673.6	710.6	794.7	825.9

Table 5: Average size and average running time of *CAmpactor* with various hyper-parameter settings of δ over all instances.

	$\delta = 5$	$\delta = 10$	$\delta = 15$	$\delta = 20$	$\delta = 25$	$\delta = 30$
avg. size	47.7	**47.4**	48.3	49.9	51.9	54.1
avg. time	296.8	284.6	266.1	243.8	222.3	202.1

Table 6: Average size and average running time of *CAmpactor* with various hyper-parameter settings of γ over all instances.

	$\gamma = 10^2$	$\gamma = 10^3$	$\gamma = 10^4$	$\gamma = 10^5$
avg. size	73.3	55.0	47.4	**43.7**
avg. time	58.9	117.2	284.6	1,173.2

Table 7: Average size and average running time of *AutoCCAG*, *FastCA*, *TCA*, *Alt-A*, *Alt-F* and *Alt-T* over all instances.

	AutoCCAG	*Alt-A*	*FastCA*	*Alt-F*	*TCA*	*Alt-T*
avg. size	86.6	47.4	86.7	**47.3**	98.1	**47.3**
avg. time	377.9	611.1	357.2	591.5	52.7	293.4

are randomly selected. To save space, Table 1 does not report the comparative results on all instances. All comparative results of *CAmpactor* and its competitors on the whole instance collection are available in our public repository.[2] In addition, the average size and the average running time of *CAmpactor* and its competitors over all PCAG instances are summarized in Table 2.

As can be clearly seen from Tables 1 and 2, *CAmpactor* can generate much smaller-sized PCAs compared to existing state-of-the-art PCAG algorithms. In particular, Table 2 presents that *CAmpactor* is able to generate PCAs with around 45% smaller sizes compared to those constructed by existing state-of-the-art algorithms, indicating the superiority of *CAmpactor* over all its competitors. As recognized by the literature [25], a PCAG instance, *i.e.,* `ecos-icse11`, is challenging to be solved, and it seemed to be impossible to generate PCAs with the size smaller than 50 for this challenging instance. More encouragingly, Table 1 demonstrates that *CAmpactor* is able to generate PCAs with the average size of 48.7 for the `ecos-icse11` instance, while the average sizes of PCAs constructed by *SamplingCA*, *AutoCCAG*, *FastCA* and *TCA* for the `ecos-icse11` instance are 108.4, 91.9, 92.4 and 102.0, respectively. Our experimental results in Tables 1 and 2 confirm that *CAmpactor* substantially advances the state of the art in solving the PCAG problem.

Here we discuss the efficiency of *CAmpactor*. According to Tables 1 and 2, it is not surprising that *CAmpactor* requires more running time than *SamplingCA*, since *CAmpactor* adopts *SamplingCA* as its initialization algorithm (as described in Section 5.4). When comparing to existing state-of-the-art meta-heuristic PCAG algorithms (*i.e., AutoCCAG*, *FastCA* and *TCA*), although *CAmpactor* needs more running time than *TCA*, *CAmpactor* runs faster than *AutoCCAG* and *FastCA*. To further study the efficiency of *CAmpactor*, we denote *CAmpactor-Short* as *CAmpactor* with shorter running time: for each instance, *CAmpactor-Short*'s running time is restricted to the average running time of *TCA* for that instance. The results of *CAmpactor-Short* are also presented in Tables 1 and 2. According to Table 2, with slightly more running time of 10.6 seconds, *CAmpactor-Short* can reduce the average size of *SamplingCA*'s generated PCAs by 21.3 over all PCAG instances. As outlined in Algorithm 2, *CAmpactor* tackles the PCAG problem by iteratively solving the decision sub-problem of finding a PCA of size λ; if a λ-sized PCA is found, *CAmpactor* continues to find a PCA of size $\lambda - 1$. Intuitively, compared to solving a decision sub-problem with smaller λ, solving a decision sub-problem with larger λ is simpler and thus requires less time. Hence, it is not surprising that *CAmpactor* greatly optimizes *SamplingCA*'s output PCAs efficiently. Also, with the same running time, the average size of PCAs output by *CAmpactor-Short* is 82.7, while this number for *SamplingCA+TCA* is 98.1. Moreover, *CAmpactor-Short* still outperforms *AutoCCAG* and *FastCA*, confirming both the effectiveness and efficiency of *CAmpactor*.

6.2 Effects of Core Techniques (RQ2)

There are two core techniques of *CAmpactor*, *i.e.,* assignment-level forbidden mechanism (in Section 4.2.1) and forced patching technique (in Section 4.2.2). To study the effect of assignment-level forbidden mechanism, based on *CAmpactor* we develop two alternative versions, *i.e., Alt-1* and *Alt-2*: *Alt-1* is *CAmpactor*'s alternative version that directly works without our assignment-level forbidden mechanism, while *Alt-2* is *CAmpactor*'s alternative version that replaces the assignment-level forbidden mechanism with the cell-level tabu mechanism [43] used in *AutoCCAG*, *FastCA* and *TCA*. To analyze the effect of our forced patching technique, we remove the forced patching technique from *CAmpactor*, resulting in an alternative version of *CAmpactor* called *Alt-3*.

Table 3 shows the average size and the average running time of *CAmpactor* and all its alternative versions over all PCAG instances. From Table 3, *CAmpactor* generates smaller-sized PCAs than all its alternative versions, indicating that each core technique of *CAmpactor* greatly contributes to the performance improvement.

6.3 Impacts of Hyper-parameter Settings (RQ3)

According to Section 4, *CAmpactor* has three hyper-parameters, *i.e.,* ψ, δ and γ, recalling that ψ controls the probability of activating the forced patching technique (in Section 4.2.2), δ is used in the assignment-level forbidden mechanism (in Section 4.2.1), and γ

decides when *CAmpactor* terminates (in Section 4.1.1). Here we analyze how the settings of these three hyper-parameters impact the practical performance of *CAmpactor*.

Tables 4, 5 and 6 report the performance of *CAmpactor* with different settings of ψ, δ and γ, respectively. According to Table 4, where the value domain of ψ ranges from 0 to 1, with the increment of 0.1, in terms of average size of generated PCAs, *CAmpactor* shows robustness when $\psi \geq 0.1$. From Table 5, where the value domain of δ varies from 5 to 30, with the increment of 5, *CAmpactor* generates the smallest-sized PCAs when δ is set to 10, and *CAmpactor* shows its effectiveness when δ is around 10. Table 6, where the value domain of γ is $\{10^2, 10^3, 10^4, 10^5\}$, shows that when γ is set to a larger value (*i.e.,* the searching budget of *CAmpactor* enlarges), *CAmpactor* generates smaller-sized PCAs and meanwhile requires longer running time. Hence, *CAmpactor* is a flexible algorithm, since it can balance effectiveness and efficiency via adjusting γ.

6.4 Optimizing Other PCAG Algorithms (RQ4)

According to Section 5.4, in the preceding experiments (Tables 1–6), *CAmpactor* adopts *SamplingCA* as its initialization algorithm, and the related results present that *CAmpactor* can greatly reduce the size of PCA constructed by *SamplingCA*. Here we aim to analyze whether *CAmpactor* can optimize the PCAs output by other state-of-the-art PCAG algorithms. Hence, we integrate *AutoCCAG*, *FastCA* and *TCA* into *CAmpactor* as *CAmpactor*'s initialization algorithm (*i.e.,* using *CAmpactor* to optimize the PCAs output by *AutoCCAG*, *FastCA* and *TCA*), resulting in three new alternative versions of *CAmpactor*, *i.e., Alt-A*, *Alt-F* and *Alt-T*, respectively.

Table 7 presents the comparative results of *AutoCCAG*, *FastCA*, *TCA*, *Alt-A*, *Alt-F* and *Alt-T* on all instances. According to Table 7, the average sizes of PCAs output by *AutoCCAG*, *FastCA* and *TCA* are 86.6, 86.7 and 98.1, respectively. Encouragingly, after the optimization of *CAmpactor*, their average sizes are decreased to 47.4, 47.3 and 47.3, respectively. Also, for *AutoCCAG*, *FastCA* and *TCA*, the additional running time brought by *CAmpactor* is around 240 seconds. Hence, our evaluation demonstrates that *CAmpactor* can further optimize PCAs output by different PCAG algorithms effectively and efficiently, indicating the generality of *CAmpactor*.

6.5 Threats to Validity

There are two potential threats to validity of this work.

Representativeness of adopted instances: Our evaluation adopts a diverse collection of 124 public PCAG instances, where each instance is collected from a real-world, highly configurable system. According to Section 5.1, these adopted instances cover a broad range of numbers of options and constraints, and they have been extensively studied in plenty of recent work [5, 32, 41, 49, 50, 60, 65, 66]. Therefore, this threat can be mitigated.

Randomness of competing approaches in our experiments: There are 5 PCAG algorithms evaluated in our experiments, *i.e., CAmpactor*, *SamplingCA*, *AutoCCAG*, *FastCA* and *TCA*, all of which are randomized algorithms. For a randomized algorithm, conducting one single run per instance might not precisely evaluate its performance. Following recent studies [42, 43, 48], for each algorithm we perform 10 independent runs per instance. Moreover, as described in Section 5.4, we conduct the significance test and calculate the effect size to examine the comparative results. As a result, this potential threat can be reduced.

7 RELATED WORK

As an important topic of software testing, combinatorial interaction testing (CIT) has been widely studied. For literature reviews on CIT, readers can refer to surveys [59, 71] and books [34, 82]. Recently, CIT has also demonstrated its usefulness in testing a variety of critical systems from diverse real-world applications, including security [52], programming language [23], deep learning [13, 51], and quantum programming [74]. Pairwise testing is the most widely-used CIT technique [50], since it can be feasibly performed in practice while preserving strong ability to detect faults [14, 26, 35, 70, 80]. This work aims to advance the state of the art in solving the PCAG problem, which is a core problem in pairwise testing.

There are three main classes of practical PCAG algorithms, *i.e.,* constraint-encoding algorithms (*e.g.,* [1, 4, 27, 78, 84]), greedy algorithms (*e.g.,* [8–10, 14, 33, 37–39, 72, 75, 77, 81]) and meta-heuristic algorithms (*e.g.,* [8, 15–17, 20, 22, 24, 30, 42, 43, 48, 55]). Given a PCAG instance, constraint-encoding PCAG algorithms first encode it into an instance of other combinatorial optimization problems, and then existing optimization solvers are used to solve the encoded instance [1, 4, 27, 78, 84]. In practice, constraint-encoding algorithms can only generate PCAs for small-scale instances. However, they cannot handle large-scale PCAG instances effectively.

For greedy PCAG algorithms, they aim to generate PCAs greedily. Greedy algorithms can be divided into two main categories, *i.e.,* one-test-at-a-time (OTAT) algorithms and in-parameter-order (IPO) algorithms. The famous *AETG* algorithm is the first OTAT algorithm [14], and since then many improvements have been proposed to strengthen its performance [8–10, 72, 78]. The IPO strategy was first introduced by Lei and Tai [39], and there exist a number of IPO's variants [33, 37, 38, 75, 81]. Among these IPO's variants, *ACTS* is a powerful IPO algorithm, and has been widely utilized in both academia and industry [81]. Although greedy algorithms can generate PCAs for medium-scale PCAG instances, their generated PCAs are of large scale. Thus, greedy algorithms are not applicable in such application scenarios with limited time budget.

Compared to constraint-encoding algorithms and greedy algorithms, meta-heuristic PCAG algorithms can generate PCAs of significantly smaller sizes yet require much longer running time. Meta-heuristic algorithms iteratively search for smaller-sized PCAs via advanced searching strategies. In practice, there are various types of meta-heuristic algorithms, including simulated annealing (*e.g., CASA* [20, 21]), hyper-heuristic search (*e.g., HHSA* [30]) and local search (*e.g., TCA*[43], *FastCA*[42] and *AutoCCAG* [48]). Unfortunately, existing PCAG algorithms (including constraint-encoding algorithms, greedy algorithms and meta-heuristic algorithms) suffer from the severe scalability challenge [50, 65, 76]. That is, when dealing with highly configurable systems with thousands of options, existing PCAG algorithms require much running time to construct large-sized PCS (*i.e.,* generating test suites of large sizes for highly configurable systems), which would make testing highly configurable systems considerably inefficient.

To alleviate the scalability challenge, Luo *et al.* proposed a state-of-the-art PCAG algorithm dubbed *SamplingCA* [50], which can

generate PCAs for large-scale instances fast via effective sampling techniques. However, *SamplingCA* does not explicitly incorporate any optimization technique to reduce the size of its generated PCA, so the size can be further reduced. Since constructing small-sized PCAs is important in practice, it is urgent to design effective algorithms to generate small-sized PCAs for large-scale instances.

This work proposes *CAmpactor*, a novel and effective local search algorithm. Experiments show that *CAmpactor* can compact PCAs into much smaller-sized ones efficiently for large-scale instances. Moreover, compared to *SamplingCA*, *CAmpactor* generates PCAs of much smaller sizes, indicating that *CAmpactor* can alleviate the scalability challenge more effectively and thus it is able to significantly advance the state of the art in PCAG solving.

8 CONCLUSION

In this work, we propose a novel and effective local search PCAG algorithm named *CAmpactor* which can further alleviate the scalability challenge. In particular, *CAmpactor* incorporates two novel techniques, *i.e.,* assignment-level forbidden mechanism and forced patching technique, to enhance its performance. Our extensive experiments on 124 public instances clearly present that *CAmpactor* generates much smaller-sized PCAs than existing state-of-the-art PCAG algorithms. Moreover, our evaluation confirms the effects of all core techniques of *CAmpactor* and present that *CAmpactor* can further optimize the PCAs output by different PCAG algorithms.

9 DATA AVAILABILITY

The source code of *CAmpactor*, all PCAG instances adopted in this work and the detailed experimental results (including those of *CAmpactor*, all its competitors, and all its alternative versions on all instances) are publicly available in our public repository: **https://github.com/chuanluocs/CAmpactor**.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China under Grant 62202025 and Grant 62122078, in part by the Open Project of Xiangjiang Laboratory under Grant 22XJ03010, in part by the Natural Science Foundation of Changsha under Grant kq2202104, and in part by the High Performance Computing Center of Central South University.

REFERENCES

[1] Carlos Ansótegui, Felip Manyà, Jesus Ojeda, Josep M. Salvia, and Eduard Torres. 2022. Incomplete MaxSAT approaches for combinatorial testing. *Journal of Heuristics* 28, 4 (2022), 377–431.
[2] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer.
[3] Adrian Balint and Andreas Fröhlich. 2010. Improving Stochastic Local Search for SAT with a New Probability Distribution. In *Proceedings of SAT 2010.* Springer, 10–15.
[4] Mutsunori Banbara, Haruki Matsunaka, Naoyuki Tamura, and Katsumi Inoue. 2010. Generating Combinatorial Test Cases by Efficient SAT Encodings Suitable for CDCL SAT Solvers. In *Proceedings of LPAR 2010.* 112–126.
[5] Eduard Baranov, Axel Legay, and Kuldeep S. Meel. 2020. Baital: An Adaptive Weighted Sampling Approach for Improved t-wise Coverage. In *Proceedings of ESEC/FSE 2020.* 1114–1126.
[6] Don S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of SPLC 2005.* 7–20.
[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2009. *Handbook of Satisfiability.* Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press.
[8] Renée C. Bryce and Charles J. Colbourn. 2007. The Density Algorithm for Pairwise Interaction Testing. *Software Testing, Verification and Reliability* 17, 3 (2007), 159–182.
[9] Renée C. Bryce and Charles J. Colbourn. 2009. A Density-based Greedy Algorithm for Higher Strength Covering Arrays. *Software Testing, Verification and Reliability* 19, 1 (2009), 37–53.
[10] Renée C. Bryce, Charles J. Colbourn, and Myra B. Cohen. 2005. A Framework of Greedy Methods for Constructing Interaction Test Suites. In *Proceedings of ICSE 2005.* 146–155.
[11] Shaowei Cai and Kaile Su. 2013. Local search for Boolean Satisfiability with configuration checking and subscore. *Artificial Intelligence* 204 (2013), 75–98.
[12] Xinye Cai, Haoran Sun, Qingfu Zhang, and Yuhua Huang. 2019. A Grid Weighted Sum Pareto Local Search for Combinatorial Multi and Many-Objective Optimization. *IEEE Transactions on Cybernetics* 49, 9 (2019), 3586–3598.
[13] Yanshan Chen, Ziyuan Wang, Dong Wang, Chunrong Fang, and Zhenyu Chen. 2019. Variable Strength Combinatorial Testing for Deep Neural Networks. In *Proceedings of ICST Workshops 2019.* 281–284.
[14] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering* 23, 7 (1997), 437–444.
[15] Myra B. Cohen, Charles J. Colbourn, and Alan C. H. Ling. 2003. Augmenting Simulated Annealing to Build Interaction Test Suites. In *Proceedings of ISSRE 2003.* 394–405.
[16] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. 2003. Constructing Test Suites for Interaction Testing. In *Proceedings ICSE 2003.* 38–48.
[17] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, Charles J. Colbourn, and James S. Collofello. 2003. A Variable Strength Interaction Testing of Components. In *Proceedings of COMPAC 2003.* 413–418.
[18] W. J. Conover. 1999. *Practical Nonparametric Statistics.* Conover.
[19] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Proceedings of SAT 2003.* 502–518.
[20] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2009. An Improved Meta-Heuristic Search for Constrained Interaction Testing. In *Proceedings of International Symposium on Search Based Software Engineering 2009.* 13–22.
[21] 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.
[22] Syed A. Ghazi and Moataz A. Ahmed. 2003. Pair-wise Test Coverage using Genetic Algorithms. In *Proceedings of CEC 2003.* 1420–1424.
[23] Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2021. Do Judge a Test by its Cover - Combining Combinatorial and Property-Based Testing. In *Proceedings of ESOP 2021.* 264–291.
[24] Loreto Gonzalez-Hernandez, Nelson Rangel-Valdez, and Jose Torres-Jimenez. 2010. Construction of Mixed Covering Arrays of Variable Strength Using a Tabu Search Approach. In *Proceedings of COCOA 2010.* 51–64.
[25] 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 (2014), 650–670.
[26] Aymeric Hervieu, Dusica Marijan, Arnaud Gotlieb, and Benoit Baudry. 2016. Practical Minimization of Pairwise-covering Test Configurations using Constraint Programming. *Information and Software Technology* 71 (2016), 129–146.
[27] Brahim Hnich, Steven David Prestwich, Evgeny Selensky, and Barbara M. Smith. 2006. Constraint Models for the Covering Test Problem. *Constraints* 11, 2-3 (2006), 199–219.
[28] Holger H. Hoos. 2002. An Adaptive Noise Mechanism for WalkSAT. In *Proceedings of AAAI 2002.* 655–660.
[29] Holger H. Hoos and Thomas Stützle. 2004. *Stochastic Local Search: Foundations & Applications.* Elsevier / Morgan Kaufmann.
[30] Yue Jia, Myra B. Cohen, Mark Harman, and Justyna Petke. 2015. Learning Combinatorial Interaction Test Generation Strategies Using Hyperheuristic Search. In *Proceedings of ICSE 2015.* 540–550.
[31] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-based Sampling of Software Configuration Spaces. In *Proceedings of ICSE 2019.* 1084–1094.
[32] 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 ESEC/FSE 2017.* 291–302.
[33] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: yet another sampling algorithm. In *Proceedings of VaMoS 2020.* 4:1–4:10.
[34] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2013. *Introduction to Combinatorial Testing.* CRC press.
[35] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421.

[36] Rick Kuhn, Raghu N. Kacker, Jeff Yu Lei, and Dimitris E. Simos. 2020. Input Space Coverage Matters. *Computer* 53, 1 (2020), 37–44.

[37] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 2007. IPOG: A General Strategy for T-Way Software Testing. In *Proceedings of ECBS 2007*. 549–556.

[38] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 2008. IPOG/IPOG-D: Efficient Test Generation for Multi-way Combinatorial Testing. *Software Testing, Verification and Reliability* 18, 3 (2008), 125–148.

[39] Yu Lei and Kuo-Chung Tai. 1998. In-Parameter-Order: A Test Generation Strategy for Pairwise Testing. In *Proceedings of HASE 1998*. 254–261.

[40] Chu Min Li and Wenqi Huang. 2005. Diversification and Determinism in Local Search for Satisfiability. In *Proceedings of SAT 2005*. 158–172.

[41] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. SAT-based Analysis of Large Real-world Feature Models is Easy. In *Proceedings of SPLC 2015*, Douglas C. Schmidt (Ed.). 91–100.

[42] Jinkun Lin, Shaowei Cai, Chuan Luo, Qingwei Lin, and Hongyu Zhang. 2019. Towards More Efficient Meta-heuristic Algorithms for Combinatorial Test Generation. In *Proceedings of ESEC/FSE 2019*. 212–222.

[43] Jinkun Lin, Chuan Luo, Shaowei Cai, Kaile Su, Dan Hao, and Lu Zhang. 2015. TCA: An Efficient Two-Mode Meta-Heuristic Algorithm for Combinatorial Test Generation. In *Proceedings of ASE 2015*. 494–505.

[44] Chuan Luo, Shaowei Cai, Kaile Su, and Wei Wu. 2015. Clause States Based Configuration Checking in Local Search for Satisfiability. *IEEE Transactions on Cybernetics* 45, 5 (2015), 1014–1027.

[45] Chuan Luo, Shaowei Cai, Wei Wu, Zhong Jie, and Kaile Su. 2015. CCLS: An Efficient Local Search Algorithm for Weighted Maximum Satisfiability. *IEEE Transactions on Computers* 64, 7 (2015), 1830–1843.

[46] Chuan Luo, Shaowei Cai, Wei Wu, and Kaile Su. 2014. Double Configuration Checking in Stochastic Local Search for Satisfiability. In *Proceedings of AAAI 2014*. 2703–2709.

[47] Chuan Luo, Holger H. Hoos, Shaowei Cai, Qingwei Lin, Hongyu Zhang, and Dongmei Zhang. 2019. Local Search with Efficient Automatic Configuration for Minimum Vertex Cover. In *Proceedings of IJCAI 2019*. 1297–1304.

[48] Chuan Luo, Jinkun Lin, Shaowei Cai, Xin Chen, Bing He, Bo Qiao, Pu Zhao, Qingwei Lin, Hongyu Zhang, Wei Wu, Saravanakumar Rajmohan, and Dongmei Zhang. 2021. AutoCCAG: An Automated Approach to Constrained Covering Array Generation. In *Proceedings of ICSE 2021*. 201–212.

[49] 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 ESEC/FSE 2021*. 1081–1092.

[50] Chuan Luo, Qiyuan Zhao, Shaowei Cai, Hongyu Zhang, and Chunming Hu. 2022. SamplingCA: Effective and Efficient Sampling-Based Pairwise Testing for Highly Configurable Software Systems. In *Proceedings of ESEC/FSE 2022*. 1185–1197.

[51] Lei Ma, Felix Juefei-Xu, Minhui Xue, Bo Li, Li Li, Yang Liu, and Jianjun Zhao. 2019. DeepCT: Tomographic Combinatorial Testing for Deep Learning Systems. In *Proceedings of SANER 2019*. 614–618.

[52] Marcel Maehren, Philipp Nieting, Sven Hebrok, Robert Merget, Juraj Somorovsky, and Jörg Schwenk. 2022. TLS-Anvil: Adapting Combinatorial Testing for TLS Libraries. In *Proceedings of USENIX Security 2022*. 215–232.

[53] Michalis Mavrovouniotis, Felipe Martins Müller, and Shengxiang Yang. 2017. Ant Colony Optimization With Local Search for Dynamic Traveling Salesman Problems. *IEEE Transactions on Cybernetics* 47, 7 (2017), 1743–1756.

[54] David A. McAllester, Bart Selman, and Henry A. Kautz. 1997. Evidence for Invariants in Local Search. In *Proceedings of AAAI 1997*. 321–326.

[55] James D. McCaffrey. 2009. Generation of Pairwise Test Sets Using a Genetic Algorithm. In *Proceedings of COMPSAC 2009*. 626–631.

[56] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of ICSE 2016*. 643–654.

[57] Hanefi Mercan, Cemal Yilmaz, and Kamer Kaya. 2019. CHiP: A Configurable Hybrid Parallel Covering Array Constructor. *IEEE Transactions on Software Engineering* 45, 12 (2019), 1270–1291.

[58] Wil Michiels, Emile H. L. Aarts, and Jan H. M. Korst. 2007. *Theoretical aspects of local search*. Springer.

[59] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *Comput. Surveys* 43, 2 (2011), 11:1–11:29.

[60] Jeho Oh, Paul Gazzillo, and Don S. Batory. 2019. t-wise Coverage by Uniform Sampling. In *Proceedings of SPLC 2019*. 15:1–15:4.

[61] Sebastian Oster, Florian Markert, and Philipp Ritter. 2010. Automated Incremental Pairwise Testing of Software Product Lines. In *Proceedings of SPLC 2010*. 196–210.

[62] Mingyu Park, Hoon Jang, Taejoon Byun, and Yunja Choi. 2020. Property-based testing for LG home appliances using accelerated software-in-the-loop simulation. In *Proceedings of ICSE-SEIP 2020*. 120–129.

[63] Justyna Petke, Myra B. Cohen, Mark Harman, and Shin Yoo. 2015. Practical Combinatorial Interaction Testing: Empirical Findings on Efficiency and Early Fault Detection. *IEEE Transactions on Software Engineering* 41, 9 (2015), 901–924.

[64] Justyna Petke, Shin Yoo, Myra B. Cohen, and Mark Harman. 2013. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of ESEC/FSE 2013*. 26–36.

[65] 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 SPLC 2019*. 14:1–14:6.

[66] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *Proceedings of ICST 2019*. 240–251.

[67] Federica Sarro, Mark Harman, Yue Jia, and Yuanyuan Zhang. 2018. Customer Rating Reactions Can Be Predicted Purely using App Features. In *Proceedings of RE 2018*. 76–87.

[68] Jialong Shi, Qingfu Zhang, and Jianyong Sun. 2020. PPLS/D: Parallel Pareto Local Search Based on Decomposition. *IEEE Transactions on Cybernetics* 50, 3 (2020), 1060–1071.

[69] Jing Sun, Hongyu Zhang, Yuan-Fang Li, and Hai H. Wang. 2005. Formal Semantics and Verification for Feature Modeling. In *Proceedings of ICECCS 2005*. 303–312.

[70] Kuo-Chung Tai and Yu Lei. 2002. A Test Generation Strategy for Pairwise Testing. *IEEE Transactions on Software Engineering* 28, 1 (2002), 109–111.

[71] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys* 47, 1 (2014), 6:1–6:45.

[72] Yu-Wen Tung and Wafa S. Aldiwan. 2000. Automating Test Case Generation for the New Generation Mission Software System. In *Proceedings of IEEE Aerospace Conference 2000*. 431–437.

[73] András Vargha and Harold 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.

[74] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Application of Combinatorial Testing to Quantum Programs. In *Proceedings of QRS 2021*. 179–188.

[75] Ziyuan Wang, Changhai Nie, and Baowen Xu. 2007. Generating Combinatorial Test Suite for Interaction Relationship. In *Proceedings of SOQUA 2007*. 55–61.

[76] 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.

[77] Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. 2016. Greedy Combinatorial Test Case Generation using Unsatisfiable Cores. In *Proceedings of ASE 2016*. 614–624.

[78] Akihisa Yamada, Takashi Kitamura, Cyrille Artho, Eun-Hye Choi, Yutaka Oiwa, and Armin Biere. 2015. Optimization of Combinatorial Testing by Incremental SAT Solving. In *Proceedings of ICST 2015*. 1–10.

[79] Cemal Yilmaz, Myra B. Cohen, and Adam A. Porter. 2006. Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces. *IEEE Transactions on Software Engineering* 32, 1 (2006), 20–34.

[80] Cemal Yilmaz, Sandro Fouché, Myra B. Cohen, Adam A. Porter, Gülsen Demiröz, and Ugur Koc. 2014. Moving Forward with Combinatorial Interaction Testing. *Computer* 47, 2 (2014), 37–45.

[81] Linbin Yu, Yu Lei, Mehra Nouroz Borazjany, Raghu Kacker, and D. Richard Kuhn. 2013. An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation. In *Proceedings of ICST 2013*. 242–251.

[82] Jian Zhang, Zhiqiang Zhang, and Feifei Ma. 2014. *Automatic Generation of Combinatorial Test Data*. Springer.

[83] Xindi Zhang, Bohan Li, Shaowei Cai, and Yiyuan Wang. 2021. Efficient Local Search based on Dynamic Connectivity Maintenance for Minimum Connected Dominating Set. *Journal of Artificial Intelligence Research* 71 (2021), 89–119.

[84] Zhiqiang Zhang, Jun Yan, Yong Zhao, and Jian Zhang. 2014. Generating Combinatorial Test Suite using Combinatorial Optimization. *Journal of Systems and Software* 98 (2014), 191–207.