Available online at www.sciencedirect.com
ScienceDirect
Applied Soft Computing 00 (2016) 1–20
Soft Computlogo
Applied Soft Computing

IDEA: A New Indicator-based Differential Evolutionary Algorithm for Optimal Feature Selection in Software Product Line Engineering

Yinxing Xue[a], Jinghui Zhong[a], Tian Huat Tan[b], Yang Liu[a], Wentong Cai[a]

[a]*Nanyang Technological University,50 Nanyang Avenue, 639798, Singapore*
[b]*Singapore University of Technology & Design, 8 Somapah Road,487372, Singapore*

Abstract

The commercial off-the-shelf software targets a variety of customers. Thus a complicated process of software configuration is often required to attain the exactly desired functionalities before the deployment of the software. In software engineering domain, software product line engineering (SPLE) is advocated to systematically derive and maintain different product variants in a software product family. To model the configuration options, feature oriented domain analysis (FODA) is proposed to capture the commonalities and competing variabilities of the product variants in an SPL. A key challenge for deriving a new product is to find a set of features that do not have inconsistencies or conflicts, yet optimize multiple objectives (e.g., minimizing cost and maximizing number of features), which are often competing with each other. In this paper, a novel evolutionary algorithm (EA) named IDEA is proposed to efficiently search for correct (i.e., satisfying requirement constrains) solutions under conflicting objectives. In the proposed IDEA, dual populations are evolved simultaneously by using different types of evolution operators. A feedback-directed mechanism is utilized to further improve the search efficiency. Our empirical results have shown that the proposed IDEA outperforms several state-of-the-art algorithms on most case studies in terms of finding correct and non-dominated solutions.

Keywords:
Software Product Line Engineering, Competing Feature Selection, Indicator-based Differential Evolutionary Algorithm (IDEA), Indicator-based Evolutionary Algorithm (IBEA)

1. Introduction

The Software Product Line Engineering (SPLE) [1] is a new software development paradigm for a set or a family of similar products. Relying on much similarity existing among software products and relevant development process [2], SPLE improves software productivity and quality. The basic idea of SPLE is to maintain a set of core assets in a reuse-based way. In last two decades, SPLE has been an active research area in software engineering [3, 4]. The motivation of SPLE originates from the various customer requirements — in reality, most of the time companies develop and maintain multiple variants of the same software for different customers. All these product variants are similar yet different in customer-specific features. These motivate the reuse of features. Systematic reuse avoids repetitions, which helps to reduce the development/maintenance effort, shorten the time-to-market and improve the overall quality of software [5].

In software family or Software Product Line (SPL), *feature model* [1] is widely used to model commonalities and competing variabilities among similar yet different products. Here, in the view of the internal developers, a *feature* is often defined as a program function which realizes a group of individual relevant requirements [6]. In the view of external customers, a feature is usually defined as a visible value, quality, or characteristic of software for the end-users [7, 8]. Based on the feature model, different features are carefully selected to meet the requirements of customers and to avoid possible conflicts or compatibility problems. In the era of a thriving market of mobile and service-based applications, vendors are required to reconfigure their applications promptly, to extend their customer base. Therefore, it is desirable to automatically derive products that satisfy various customers and avoid feature conflicts.

A *feature model* provides a representation of functionality in an SPL, which could be used to facilitate the reasoning and configuration of the SPL [8]. Common SPLs consist of hundreds or even thousands of features. For instance, as reported in [9], the Linux X86 kernel contains 6888 features, and 343944 constraints. In addition, the features are usually associated with quality attributes such as cost and reliability. This complexity provides challenges for the reasoning and configuration of feature models. It is hard for the vendor to select a set of features that complies with the feature model, and meanwhile optimizes the quality attributes according to user preferences. This is called the *optimal feature selection* problem [10].

Existing works [10, 11, 12, 13] have adopted evolutionary algorithms (EAs) for feature selection with resource constraints and product generation based on the value of user preferences, respectively. Guo *et al.* [10] proposed a genetic algorithm (GA) approach for tackling the optimal feature selection problem. In their work, a repair operator is used to fix each candidate solution, so that it is fully compatible with the feature model after each round of crossover and mutation operations. This approach might be non-terminating, and furthermore, it does not take advantage of the automatic correction that is brought by the GA. In addition, GA combines all objectives into a single fitness function with respective weights. This only gives users a solution that is specific to the weights used in the objective formula.

To address this problem, Sayyad *et al.* [11, 12] proposed an approach that uses EAs to support multi-objective optimization, and a range of optimal solutions (i.e., a Pareto front) is returned to the user as a result. They investigated seven EAs and discovered that the Indicator-Based Evolutionary Algorithm (IBEA) [14] yields the best results among the seven tested EAs in terms of time, correctness (i.e., the validity of the product in satisfying requirement constraints) and satisfaction to user preferences. After that, several approaches are proposed to enhance the IBEA. In [13], they used a static method to prune features before execution of IBEA for reducing search space. They also introduced a "seeding method" by pre-computing a correct solution, which was subsequently used by IBEA to generate further correct solutions.

In [15], to improve the correctness of the found solutions, we introduce a novel feedback-directed mechanism to existing EAs. In our approach, the feature model is first preprocessed based on SAT solving to remove the prunable features, before the execution of an EA. We have shown that we always prune more features compared to the pruning method in [13]. During each round of executing EA, the violated constraints would be analyzed. The analyzed results are used as feedback to guide evolutionary operators (i.e., crossover and mutation) for producing offsprings for the next round.

Existing EA-based approaches mainly focus on finding correct solutions and IBEA is one of the most powerful approaches that can obtain solutions with high correctness ratio [15]. However, IBEA will produce many duplicated solutions in the evolving population. To address this issue, in this paper, we extend the IBEA with the idea of differential evolution for the purpose of finding more *unique* and *non-dominated* correct solutions.

The key idea of our approach, namely, IDEA (Indicator-based Differential Evolutionary Algorithm), is to evolve dual populations with different evolution mech-

anisms. The first population is evolved by using the traditional IBEA operators which was proposed in [15]. The population focuses on obtaining enough correct solutions as well as to maintain population diversity. Meanwhile, the second population is evolved by using operators extended from the differential evolution (DE). We choose DE operators to evolve the second population because these operators have been shown quite effective to maintain population diversity and very efficient to search the optimal solution [16]. The offspring generated by one population are shared with the other population so as to improve the search efficiency. To evaluate our proposed method, both SPLOT [17] and LVAT [18] repositories are utilized. SPLOT is a repository of feature models used by many researchers as a benchmark, and LVAT contains the real-world feature models which have large feature sizes, including the aforementioned Linux X86 kernel model which contains 6888 features.

Our main contributions are summarized below.

1. We introduce a new indicator-based differential evolutionary algorithm. The basic idea is to combine one IBEA population and one DE population to achieve both the correct and non-dominated solutions.
2. We introduce a feedback-directed mechanism into existing EAs. In a feedback-directed EA, solutions are analyzed by their violated constraints. The information is used as feedback for evolutionary operators to produce offspring that are more likely to satisfy more constraints.
3. We adopt the seeding method as proposed in [13] for finding valid solutions in the Linux X86 feature model, which contains 6888 features. Our IDEA combining with the seeding method has significantly shortened the search time, compared with the original seeding approach as proposed in [13].
4. We evaluate our approach with feature models that are available publicly. Our IDEA is able to find much more unique and non-dominated solutions than the state-of-the-art algorithm IBEA [14], meanwhile retaining a comparable correctness rate with IBEA. The feedback-directed mechanism also exhibits a significant improvement on finding more optimized valid solutions, compared with the original unguided EAs used in [11, 13].

In [15], we mainly introduce the optimization and how it can improve existing EAs. In this paper, we extend [15] by introducing the novel IDEA. We also compare IDEA with IBEA and evaluate how the optimization can improve IDEA.

Outline. Section 2 introduces the background of this work. Section 3 elaborates the steps of IDEA. Section 4 presents the feedback-directed mechanism and the feature preprocessing. Section 5 provides the evaluation of our approach. Section 6 reviews related works. Finally, Section 7 concludes and outlines future work.

2. Background

In this section, we provide the background knowledge on software product line, feature model, and multi-objective optimization problem.

2.1. Software Product Line

Software product line engineering (SPLE) is architecture-centric and feature-oriented, as SPLE adopts feature-oriented domain analysis [8] for requirements analysis and builds core assets architecture for reuse [2]. Technically, SPLE is a two-phase approach composed of domain engineering and application engineering. The task of domain engineering is to build the software product line (SPL) architecture consisting of a core-asset base and the variant features, while the application engineering focuses on derivation of new products by different customizations of variant features applied onto the core-asset base. Thus, automation of processing and verification of product derivation is a fundamental problem in SPLE. Exploring an efficient and scalable approach for the optimal feature selection problem is critical to the success of SPLE.

2.2. Feature Model and its Semantics

The concept of feature model in domain engineering is to represent the features within the product family as well as the structural and semantic (require or exclude) relationships between those features [8]. Since the proposal of SPL, feature model has even been characterized as "the greatest contribution of domain engineering to software engineering" [19].

A feature model is a tree-like hierarchy of features. The structural and semantic relationships between a parent (or compound) feature and its child features (or subfeatures) can be specified as:

- *Alternative* – If the parent feature is selected, exactly one among the exclusive subfeatures should be selected,
- *Or* – If the parent feature is selected, at least one of the subfeatures must be selected,
- *Mandatory* – A mandatory feature must be selected if its parent is selected,

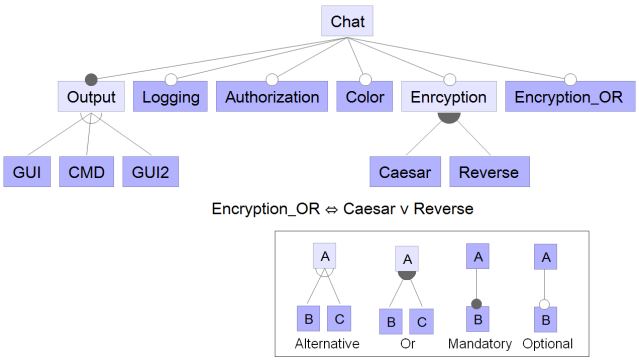

Figure 1. The feature model of *JCS*

- *Optional* – An optional feature is optional to be selected.

Besides the above structure or parental relationships between features, cross-tree constraints (CTCs) are also often adopted to represent the mutual relationship for features across the feature model. There are three types of common CTCs:

- f_a *requires* f_b – The inclusion of feature f_a implies the inclusion of feature f_b in the same product.
- f_a *excludes* f_b – The inclusion of feature f_a implies the exclusion of feature f_b in the same product, and vice versa.
- f_a *iff* f_b – The inclusion of feature f_a implies the inclusion of feature f_b in the same product, and vice versa.

In Figure 1, the feature model of a Java Chat System (*JCS*) is illustrated. The root feature of the feature model is *Chat*, which has a mandatory subfeature (*Output*) and several optional subfeatures (e.g., *Encryption*). Since the feature *Output* is mandatory, exactly one of its subfeatures (*GUI*, *CMD*, and *GUI2*) must be selected. In addition, if the *Encryption* feature is selected, at least one of its subfeatures (*Caesar* and *Reverse*) needs to be selected. There is a CTC for *JCS* which is of the form f_a *iff* f_b – *Encryption_OR* is selected if and only if *Caesar* or *Reverse* is selected.

The feature model listed in Figure 1 can be captured by the constraints that are listed in Table 1. The constraints are specified according to the semantics of feature model. Constraint c(1) specifies that the root feature must be present, to prevent a trivial feature model with no selected feature. Constraint c(2) specifies the mandatory feature *Output* and constraints c(3) – c(7) specify constraints on the other five optional subfeatures. The subfeatures of *Output* are in an *Alternative* relationship.

Table 1. Constraints of *JCS*

$Chat$	c(1)
$Output \iff Chat$	c(2)
$Logging \implies Chat$	c(3)
$Authorization \implies Chat$	c(4)
$Color \implies Chat$	c(5)
$Encryption \implies Chat$	c(6)
$Encryption\_OR \implies Chat$	c(7)
$(GUI \vee CMD \vee GUI2) \iff Output$	c(8)
$\neg(GUI \wedge CMD)$	c(9)
$\neg(GUI \wedge GUI2)$	c(10)
$\neg(CMD \wedge GUI2)$	c(11)
$(Caesar \vee Reverse) \iff Encryption$	c(12)
$Encryption\_OR \iff (Caesar \vee Reverse)$	c(13)

This is specified using constraints c(8) – c(11). Constraint c(8) states that *Output* is selected, if and only if at least one of *CMD*, *GUI* and *GUI2* is selected. Constraints c(9) – c(11) specify that at most one feature from *CMD*, *GUI* and *GUI2* can be chosen. The subfeatures of *Encryption* are in *Or* relationship. The constraint c(12) denotes if *Encryption* is selected, then at least one feature from *Caesar* and *Reverse* needs to be selected, and vice versa. The only CTC of *JCS* is captured in the constraint c(13). Constraints c(1) – c(12) are called *tree constraints*, since they are related to the tree structure of the feature model. Henceforth, given a feature model M, we simply refer tree constraints and CTCs of the M, as the *constraints* of M. We denote the conjunction of constraints of M as $conj(M)$. We use $Fea(M)$ to denote the set of entire features of the feature model M. For the *JCS* example, $Fea(JCS) = \{Chat, \ldots\}$ and $|Fea(JCS)|$=12.

Definition 1 (Feasible feature set). *Given a feature model M, a* feasible feature set *for M is a non-empty feature set $F \subseteq Fea(M)$, such that F satisfies the constraints of M.*

We write $F \models M$ if $F \subseteq Fea(M)$ is a feasible feature set of the feature model M.

Example. We use *JCS* as an example. $F = \{Chat, Output, GUI\}$ is a feasible feature set of *JCS*, i.e., $F \models JCS$.

2.3. Multi-objective Optimization Problem

Many real-world problems have multiple objectives that need to be optimized simultaneously. However, these objectives usually conflict with each other, which prevents optimizing all objectives simultaneously. A remedy is to have a set of optimal trade-offs between the conflicting objectives.

A k-objective optimization problem could be written in the following form:

$$Minimize\ Obj(F) = (Obj_1(F), Obj_2(F), ..., Obj_k(F)) \quad subject\ to\ F \models M \tag{1}$$

where $Obj(F)$ is a k-dimensional objective vector for F and $Obj_i(F)$ is the value of F for ith objective.

Given $F_1, F_2 \models M$, F_1 can be viewed as better than F_2 for the minimization problem in Equation (1), if Equation (2) holds.

$$\forall i : Obj_i(F_1) \leq Obj_i(F_2) \wedge \exists j : Obj_j(F_1) < Obj_j(F_2) \tag{2}$$

where $i, j \in \{1, \ldots, k\}$.

In such a case, we say that F_1 *dominates* F_2. F_1 is called a *Pareto-optimal solution* if F_1 is not dominated by any other $F \models M$. We denote all Pareto-optimal solutions as the *Pareto front*.

Many evolutionary algorithms (e.g., IBEA [14], NSGA-II [20], ssNSGA-II [21], MOCell [22]) are proposed to find a set of *non-dominated solutions* that approximate the Pareto front for solving the multi-objective optimization problem.

Problem Statement. Our work addresses the *optimal feature selection*, which aims at searching for feasible feature sets that approximate the Pareto front to solve the multi-objective optimization problem.

3. Indicator-based Differential Evolution

This section describes the proposed methodology, which is an enhanced version of the IBEA, to address the optimal feature selection problem. First, a brief introduction on evolutionary algorithm (EA) and IBEA are given. Then, the general framework of our proposed method and the implementation of each step of the framework are presented.

3.1. Preliminaries of Evolutionary Algorithm

Evolutionary algorithms (EAs), inspired by the "survival of the fittest" principle of the Darwinian theory of natural evolution, are stochastic search methods based on principles of the biological evolution. By applying the EA, a problem is encoded into a simple chromosome-like data structure, and then evolutionary operators (e.g., selection, crossover, and mutation) are applied on these data structures to preserve "the fittest" information, which is analogous to "survival of the fittest" in the natural world. EAs often perform well in approximating solutions, and therefore EAs are typically suitable for the optimization problems especially if the search space of the problem is large and complex.

Algorithm 1: The common procedure of EA

```
1 Step-1: Initialization
  /* Generate an initial population P and
     evaluate every individual in P.        */
2 while stopping conditions are not satisfied do
3     i. Step-2:Selection
      /* Select individuals from P to form a
         temporary population P'.            */
4     ii. Step-3:Population production
      /* Perform genetic operators (e.g.,
         crossover and mutation) on P' to
         generate a new population P.        */
```

The common procedure of EAs consists of three main steps, as described in Algorithm 1. The first step is to generate an initial population of chromosomes (labeled as P), each of which represents a candidate solution of the problem and is randomly generated. Then the second step adopts the specific selection strategy to select chromosomes from P to form a temporary population P'. After that, the third step applies genetic operators such as crossover and mutation on P' to generate the new population P. The second step and the third step are repeated until the termination condition is met. An example of the termination condition might be that the number of generations exceeds a predefined upper bound $n \in \mathbb{Z}_{>0}$.

IBEA is a new EA paradigm for solving multi-objective optimization problems (MOPs). Unlike traditional multi-objective evolutionary algorithms (MOEAs) that use Pareto-based fitness assignment scheme, IBEA adopts a new indicator-based measure to assign fitness values of individuals. This new fitness assignment scheme can flexibly integrate preferences of decision maker to search for trade-off solutions. Specifically, given a population P, the indicator-based measure of $x_1 \in P$ is defined as follows:

$$F(x_1) = \sum_{x_2 \in P/\{x_1\}} -e^{-I(\{x_2\},\{x_1\})/\kappa} \tag{3}$$

where κ is a scaling factor, and I is a binary quality indicator that describes the preferences of decision maker. The binary quality indicator, a function that maps m Pareto set approximations to a real number, is used to compare the quality of two Pareto set approximations relatively to each other. The commonly used binary quality indicator is the $\epsilon-$indicator $I_{\epsilon+}$ which is defined:

$$I(A, B) = min_\epsilon\{\forall x_2 \in B,\ \exists x_1 \in A : f_i(x_1) - \epsilon \leq f_i(x_2)\ \ for\ i = \{1, ..., n\}\} \tag{4}$$

Algorithm 2: The basic procedure of IBEA

```
1 Initialization
  /* Generate an initial population P and
     evaluate every individual in P.        */
2 Fitness assignment
  /* calculate fitness values of individual
     in P by (3)                            */
3 while stopping conditions are not satisfied do
4     Selection
      /* Perform binary tournament selection
         with replacement on P in order to
         fill the temporary mating pool P'.
         */
5     Population production
      /* Perform genetic operators (e.g.,
         crossover and mutation) on P' to
         generate a set of offspring newP.  */
6     Environmental selection
      /* P = P ∪ newP;                      */
      /* Iteratively update fitness values of
         individual in P and remove the worse
         individual until the size of P does
         not exceed NP.                     */
```

where A and B are two Pareto set approximations, f_i is the ith objective function, and n is the total number of objectives. (Here we assume that the task is to minimize all objective values.) Based on the above indicator-based fitness assignment scheme, the basic procedure of IBEA is described in Algorithm 2.

3.2. Genetic Encoding of the Feature Set

The selected features of a feature model is encoded using an array-based chromosome as shown in Figure 3. Given a chromosome of length n, array indices are numbered from 0 to $n-1$. Each feature is assigned with an array index starting from 0. Each value on the chromosome ranges over $\{0, 1\}$, where 0 (resp. 1) represents the absence (resp. presence) of the feature. Given a feature model M, we define a function $f_M : Fea(M) \rightarrow \{\mathbb{Z}, \perp\}$ that maps each feature f of the feature model M to an array index. $f_M(f_1) = \perp$ denotes that there is no array index that is assigned for the feature f_1. Similarly, we define ${f_M}^{-1} : \mathbb{Z} \rightarrow Fea(M)$ as a function that maps a given array index to the feature it represents.

3.3. The Proposed Methodology

The IBEA has shown great potential to find high quality solutions for the SPLE problem. However, a major drawback of the IBEA is that it often provides many duplicated non-dominated solutions at the end of the algorithms. In order to find better and more distinct non-dominated solutions, we propose an enhanced methodology, namely, the indicator-based differential evolution (IDEA) for solving the SPLE problem. The key idea is to incorporate multiple genetic operators to evolve dual populations simultaneously, so as to maintain population diversity and lead to more non-dominated solutions. The procedure of the proposed methodology is illustrated in Figure 2. The implementations of the major steps are described below.

3.3.1. Initialization

The first step is to generate two random populations, namely *population*1 and *population*2. Each population contains NP random chromosomes, with each represented by a vector of integers:

$$X_i = [x_{i,1}, x_{i,2}, \ldots, x_{i,n}] \tag{5}$$

where X_i represents the ith chromosome in the first (or second) population, and n is the number of features under consideration. The value of $x_{i,j}$ can be 0 or 1. If $x_{i,j} = 1$, the jth feature is selected, otherwise, the jth feature is ignored. The value of $x_{i,j}$ is randomly initialized in the first generation.

3.3.2. Update population1 using IBEA operators

This step updates *population*1 using the operators of IBEA proposed in [15]. Generally, there are three sub-steps to achieve the goal of this step. In the first sub-step, the selection, crossover and mutation operators in IBEA are utilized to generate $NP/2$ offspring. Then in the second sub-step, the offspring generated by the IBEA operators and those generated by the *population*2 are join *population*1 to form a new population. Note that the size of the new population now may be larger than NP. In the third sub-step, the *Environmental selection* is utilized to remove some individuals in new population (P_1). In the *Environmental selection*, the fitness values of chromosomes in the new population are evaluated using the indicator-based approach. Then the worse chromosome in P_1 are removed one-by-one until the size of the new population does not exceed NP, as done in traditional IBEA. The revised P_1 is then used to replace *population*1 for the next generation.

3.3.3. Update population2 using DE operators

In this step, the DE operators are utilized to update *population*2. This step contains four sub-steps as described below.

- DE mutation: This sub-step aims to perform DE mutations on each chromosome in *population*2 to

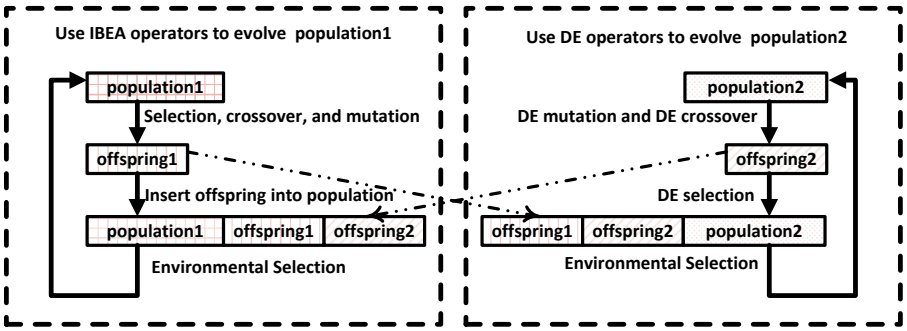

Figure 2. The general procedure of the proposed IDEA.

generate a set of mutant vectors. We adopt the "DE/current-to-rand/2" mutation scheme to accomplish task, due to its strong global search ability. The "DE/current-to-rand/2" mutation can be expressed by

$$Y_i = X_{r1} + F \cdot (X_{r2} - X_{r3}) + F \cdot (X_{r4} - X_{r5}). \quad (6)$$

where $r1$, $r2$, $r3$, $r4$, and $r5$ are five distinct random individual indices, and F is a scaling factor. Since the traditional "DE/current-to-rand/2" mutation is only applicable to real coded chromosomes, we adopt the strategy proposed in [23] to extend the 'DE/current-to-rand/2" mutation to integer-coded chromosomes. Specifically, for each element $(x_{i,j})$ of the ith parent chromosome, a mutation probability is calculated by:

$$\begin{aligned}\varphi =& F \cdot \psi(x_{r1,j}, x_{r2,j})) + F \cdot \psi(x_{r3,j}, x_{r4,j})) \\ & - F \cdot \psi(x_{r1,j}, x_{r2,j})) * F \cdot \psi(x_{r3,j}, x_{r4,j}))\end{aligned} \quad (7)$$

where $\psi(a, b)$ is defined as

$$\psi(a, b) = \begin{cases} 1, \text{if } a \neq b \\ 0, \text{otherwise} \end{cases} \quad (8)$$

The element $x_{i,j}$ will have a probability of φ to be mutate to a new value. When a mutation is required, the new value to assign $x_{i,j}$ is randomly selected from {0, 1},otherwise, its value is set equal to $x_{r1,j}$.

In order to improve the probability of generating feasible solutions, we adopt a tournament selection (with tournament size = T) to select i and $r1$, so that correct solutions (i.e., those where no constraint is violated) is more likely to be selected. What we hope is that the building blocks selected from the correct solutions should be more likely to build new correct solutions. In addition, to improve the robustness of the algorithm, the values of F is randomly set by:

$$F = rand(0, 1) \quad (9)$$

where $rand(a, b)$ returns a random value uniformly distributed within $[a, b]$.

- DE crossover: In this sub-step, each selected parent chromosome X_i is crossover with its mutant vector Y_i to generate a trial vector U_i:

$$u_{i,j} = \begin{cases} y_{i,j}, \text{if} \quad rand(0, 1) < CR \quad \text{or} \quad j = k \\ x_{i,j}, \text{otherwise} \end{cases} \quad (10)$$

  where CR is the crossover rate, k is a random integer between 1 and n, $u_{i,j}, y_{i,j}$ and $x_{i,j}$ are the jth variables of U_i, Y_i and X_i respectively. Similar to F, the value of CR is set to be $rand(0, 1)$ so as to improve the robustness of the algorithm.

- DE selection: This step adopts the one-to-one selection strategy in DE to insert newly created trial vectors into *population*2. Then for each X_i in *population*2, if X_i is dominated by U_i, then X_i is replaced by U_i, otherwise, if X_i and U_i are non-dominate with each other, then U_i is added to the tail of *population*2. The offspring generated by the first population are also added to the tail of *population*2.

- Environmental selection: The goal of this sub-step is to remove some worse individuals in *population*2 to form a new *population*2 for the next generation. Firstly, the fitness values of all individuals in *population*2 are evaluated by (3). Then the worse chromosome in *population*2 are removed one-by-one until the size of *population*2 does not exceed NP, as done in traditional IBEA.

The *population*1 and *population*2 are updated iteratively by using the above two mechanisms until the termination condition is met. When the termination condition is met, the two populations are joint to form an archive. Then the *environmental selection* is performed on the archive to select NP best individuals as the final

Algorithm 3: PrunableFeatures

input : Feature model M
output: Common features $F_c \subseteq Fea(M)$
output: Dead features $F_d \subseteq Fea(M)$

1 $F_c \leftarrow \emptyset$;
2 $F_d \leftarrow \emptyset$;
3 **foreach** $f \in Fea(M)$ **do**
4 **if** $\neg SAT(conj(M) \wedge \neg f)$ **then**
5 $F_c = F_c \cup f$;
6 **else if** $\neg SAT(conj(M) \wedge f)$ **then**
7 $F_d = F_d \cup f$;
8 **return** (F_c, F_d);

approximation Pareto optimal solutions (i.e., the alternative trade-off solutions). It should be noted that DE is a popular and powerful EA for global optimization. The operators of DE have been shown quite effective to maintain population diversity and find global (or near global) optimal solutions [16]. Meanwhile, the operators of IBEA has been shown to be capable of generating high-quality non-dominated solutions. In the proposed IDEA, both the operators of DE and IBEA are integrated to generate offspring simultaneously. By doing so, we hope to improve the population diversity, and obtain better and more none-dominated solutions.

4. Optimization

In this section, we elaborate our approach in addressing the optimal feature selection problem. First, we introduce a preprocessing method to filter out prunable features before the execution of an EA, in order to reduce the search space. Second, we illustrate feedback-directed evolutionary operators that are used in this work to guide an EA for the optimal feature selection.

4.1. Preprocessing of Feature Model

In the following, we introduce the features that could be pruned from $Fea(M)$ before the execution of an EA. By doing this, the search space of the EA would be reduced, which could make the optimal feature selection more efficient.

Our approach of preprocessing is by exploiting the *commonalities* [24] of the products. Observed that some features must be present in all products derived from M. For example in *JCS*, the feature set $\{Chat, Output\}$ is shared by all derived products, and we call these features as *common features*. Similarly, we call the set of features

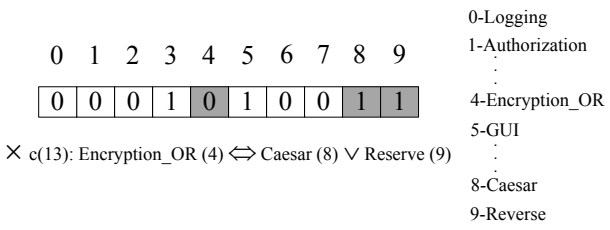

Figure 3. Feedback-directed mutation operator

that must not be used in all derived products as *dead features*. Dead features do not present in *JCS* but they are common in feature models of real systems (e.g., Linux X86 kernel and eCos operating system). Henceforth, we denote common features and dead features as F_c and F_d respectively, where $F_c, F_d \subseteq Fea(M)$, and $F_c \cap F_d = \emptyset$. The preprocessed features that are passed to the execution of EAs is $Fea(M) \setminus (F_c \cup F_d)$, and we denote $F_c \cup F_d$ as *prunable features*.

The function *PrunableFeatures* (Algorithm 3) is used to find common and dead features. Recall that $conj(M)$ represents the conjunction of all tree constraints and CTCs of feature model M, and *SAT* is a function that is used to check the satisfiability of the constraints. Note that *SAT* function is readily provided by many off-the-shelf SAT solvers (e.g., SAT4J [25]). We assume that $conj(M)$ is satisfiable, i.e., there exists at least a valid product from the feature model M. If $conj(M) \wedge \neg f$ is unsatisfiable (line 4), it implies that feature f must exist in all derived products of M. Therefore, feature f is added to common features F_c (line 5). This is similar to the detection of dead features in lines 6 and 7.

Example. We show how a feature set on the *JCS* is encoded. Note that features *Chat* and *Output* have been pruned by the preprocessing algorithm in Algorithm 3; therefore, they are not contained in the chromosome (i.e., $f_M(Chat) = f_M(Output) = \perp$). The features are indexed level by level, and their indexes have been listed in Figure 3 (e.g., $f_M(Logging) = 0$). The chromosome in Figure 3 represents the feature set $\{Encryption, GUI, Caesar, Reverse\}$.

4.2. Feedback-Directed Evolutionary Operators

The violated constraints of a chromosome C_i provide an important clue on which features on the chromosome C_i need to be modified. If we focus on these features, we may converge faster on the optimal feature selection.

We incorporate this feedback into the crossover and mutation operations, which are the main evolutionary operators common for almost all EAs. The feedback-directed crossover and mutation operators provide an

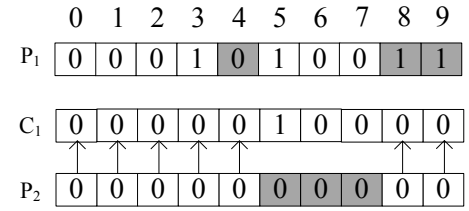

Figure 4. Feedback-directed crossover operator

Algorithm 4: FMutation

input : Chromosome P
input : Error mutation probability P_{emut}
input : Mutation probability P_{mut}
output : Chromosome C

1 $C \leftarrow P$;
2 $n \leftarrow |P|$;
3 $Err \leftarrow ErrPos(C)$;
4 **for** $i = 0$ **to** $n - 1$ **do**
5 **if** $(i \in Err \wedge rand(0,1) < P_{emut}) \vee$
6 $(i \notin Err \wedge rand(0,1) < P_{mut})$ **then**
7 $C[i] \leftarrow randInt(0,1)$;
8 **return** C;

effective guidance for EAs to perform the optimal feature selection.

4.2.1. Feedback-Directed Mutation

The objective of *mutation* operator is to change some values in a selected chromosome leading to additional genetic diversity to help the search process escape from local optimal traps.

We introduce how the *feedback-directed mutation* operator works. Before the mutation, the feedback-directed mutation analyzes the selected chromosome on the violated constraints. We denote the corresponding positions on the chromosomes for the features that are contained in the violated constraints as *error positions*.

Example. We illustrate the feedback-directed mutation operator, using the *JCS* example shown in Figure 3. Given the values of the chromosome as shown in Figure 3, we can easily check that it violates the constraint c_{13}. The constraint c_{13} contains three features, which are *Encryption_OR*, *Caesar*, and *Reverse*. The corresponding array positions of these three features are shaded on the chromosome in Figure 3. These shaded positions are the *error positions*.

The algorithm *FMutation* for feedback-directed mutation are given in Algorithm 4. At line 1, an offspring chromosome C is initialized with values in the chromosome P, and $n \in \mathbb{Z}$ is initialized with the length of the chromosome P (line 2). At line 3, $Err \in \mathcal{P}(\mathbb{Z})$ is assigned with the set of integers that is returned from $ErrPos(C)$ (which will be introduced later). The set of integers returned by $ErrPos(C)$ represents the error positions on the chromosome C. Each position on the chromosome is iterated (line 4). The function $rand(a, b)$ (resp., $randInt(a, b)$), with $a > b$, chooses a real (resp., integer) number between numbers a and b. At line 5, if the current position i is an error position, and the random number is less than the error mutation probablity P_{emut}, then the value in the ith-position on the chromosome is mutated by randomly choosing an integer between 0 and 1 (line 7). On the other hand, if the position does not belong to any error position, and the random number is less than P_{mut} (line 6), the value in the ith-position is mutated. Note that the probability P_{emut} will be set with a value that is far larger than P_{mut}, so that the mutation occurs more frequently on error positions. For P_{emut} and P_{mut}, example values could be 1.0 and 0.0000001. Note that we set P_{mut} much lower than classic mutation probability (e.g. 0.001-0.05 [26]). This is because lower P_{mut} with higher P_{emut} would lead to faster convergence, since it allows faster correction of constraint violations by minimizing the changes of non-error positions and focusing on the changes of error positions. This is demonstrated in Section 5.3.

Algorithm 5: ErrPos

input : Chromosome C
input : A set of constraints *constraints*
output : A set of integers *ePos*

1 $ePos \leftarrow \emptyset$;
2 $n \leftarrow |C_1|$;
3 $\Pi \leftarrow \emptyset$;
4 **for** $i = 0$ **to** $n - 1$ **do**
5 $\Pi \leftarrow \Pi \cup \{{f_M}^{-1}(i) \mapsto (C[i] \neq 0)\}$;
6 **foreach** *feature* $\in F_c$ **do**
7 $\Pi \leftarrow \Pi \cup \{F_c \mapsto true\}$;
8 **foreach** *feature* $\in F_d$ **do**
9 $\Pi \leftarrow \Pi \cup \{F_d \mapsto false\}$;
10 **foreach** *constraint* $\in$ *constraints* **do**
11 **if** $\Pi \not\models$ *constraint* **then**
12 $ePos \leftarrow ePos \cup getFeatures(constraint)$;
13 **return** $ePos$;

We now introduce the *ErrPos* function described in Al-

gorithm 5. At line 1, *ePos* is initialized with an empty set. The valuation function $\Pi : Fea(M) \rightarrow \{true, false\}$ (line 3) maps each feature f of the feature model M to a Boolean value that denotes whether the corresponding feature is selected. The mappings in Π are populated according to the values on the chromosome (line 5). Subsequently, common and dead features are added to the mappings in Π with values *true* and *false* respectively (lines 7–9). The reason is that common (dead resp.) features must (must not resp.) belong to any feasible feature set of feature model M as explained in Section 4.1. At line 11, $\Pi \not\models constraint$ holds iff replacing each feature f contained in the *constraint* with $\Pi(f)$ evaluates to false. In other words, $\Pi \not\models constraint$ means that the selection represented by chromosome C violates the constraint *constraint*. In such a case, the function *getFeatures*(c) is used to get the features that are contained in the constraint c (line 12). For example, given the constraint $c(13)$ in Table 1 for *JCS* as an input, *getFeatures* will return $\{4, 8, 9\}$. These array indexes that represent the error positions will be included in *ePos*.

4.2.2. Feedback-Directed Crossover

The crossover operation is used to generate offspring by exchanging values in a pair of chromosomes chosen from the population, and it happens with a probability P_{cross} (the crossover probability). The feedback-directed crossover operator uses values in the non-error positions to crossover. The objective for using values from non-error positions is to pass the "good genes" to offspring.

Example. We demonstrate the feedback-directed crossover operator, using the *JCS* example shown in Figure 4. Suppose the chromosomes P_1 and P_2 have violated constraints $c(13)$ and $c(8)$ respectively. The offspring chromosome C_1 is first initialized as the same values with the chromosome P_1. We now show that how the feedback-directed crossover is performed. The values from non-error positions of the chromosome P_2 are copied to the chromosome C_1 (shown by the arrows). This results in the chromosome C_1 that is shown in Figure 4. The production of the chromosome C_2 (not shown in the graph) is symmetric to the production of the chromosome C_1.

The algorithm *FCrossover* of feedback-directed crossover operator is given in Algorithm 6. The chromosomes C_1 and C_2 are initialized with the values from chromosomes P_1 and P_2 respectively (lines 1, 2). If the generated random number is smaller than the crossover probability P_{cross} (line 4), then it will perform the crossover operation. First, it verifies whether there exists any error position in chromosomes P_1 and P_2, by checking whether the size of their error positions

Algorithm 6: FCrossover

```
   input  : Chromosome P1
   input  : Chromosome P2
   input  : Crossover probability P_cross
   output : Chromosomes C1, C2
1  C1 ← P1;
2  C2 ← P2;
3  n ← |P1|;
4  if rand(0, 1) < P_cross then
5      if |ErrPos(P1)| > 0 ∧ |ErrPos(P2)| > 0 then
6          for i = 0 to n − 1 do
7              if i ∉ ErrPos(P1) then
8                  C2[i] ← P1[i];
9              if i ∉ ErrPos(P2) then
10                 C1[i] ← P2[i];
11     else
12         crossIndex ← randInt(0, n−1);
13         for i = crossIndex to n−1 do
14             C1[i] ← P2[i];
15             C2[i] ← P1[i];
16 return (C1, C2);
```

is greater than 0 (line 5). If it is, then the feedback-directed crossover will be performed. The algorithm iterates through the chromosome (line 6), and copies the values of non-error positions from chromosome P_1 (resp., P_2) to the corresponding positions in chromosome C_2 (resp., C_1) (lines 7–10).

Otherwise, if both chromosomes P_1 and P_2 do not have any error position, the classic single point crossover operator is applied. First, an array index, *crossIndex* $\in \{0, \ldots, n-1\}$, is randomly selected. Subsequently, all values starting from the position *crossIndex* are copied from chromosome P_2 (resp., P_1) to chromosome C_1 (resp., C_2) (lines 12–15).

5. Evaluation

We conducted experiments to evaluate the IDEA and also the optimizations. Specifically, we attempted to answer the following research questions:

RQ1. Is the IDEA *comparable* with the best EA algorithm, namely IBEA, from the existing ones in terms of the correctness of the solutions?

RQ2. What is the *runtime* of IDEA compared to the existing state-of-the-art EAs?

RQ3. Can IDEA find *more* non-dominated solutions compared with the best EA algorithm for this problem, namely IBEA?

RQ4. How does the optimization *improve* on the existing state-of-the-art methods in terms of the correctness and the runtime?

RQ5. Can the optimization be *generalized* to different EAs?

RQ6. How *scalable* are IDEA and the optimization in terms of the size of feature models?

5.1. Experimental Setup

To conduct a fair comparison between different EAs, we use the existing EAs from the same third party library, jMetal [27]. For IDEA, we also implement it in the same way as other EAs in jMetal.

5.1.1. Implementation

We have implemented our approach based on jMetal [27], which is a Java-based open source framework that supports multi-objective optimization with EAs. Sayyad *et.al* [11, 12] have made an extensive experiments to test how different EAs implemented in jMetal could contribute to the optimal feature selection. Besides IDEA, we also use the following EAs that are reported to work well in their experiments, and evaluate how the preprocessing and feedback-directed mechanisms affect these EAs.

1. **IDEA**: our Indicator-based Differential Evolutionary Algorithm
2. **IBEA**: Indicator-Based Evolutionary Algorithm [14]
3. **NSGA-II**: Nondominated Sorting Genetic Algorithm [20]
4. **ssNSGA-II**: Steady-state NSGA-II [21]
5. **MOCell**: A Cellular Genetic Algorithm for Multiobjective Optimization [22]

A brief overview of these EAs are provided in Table 3.

5.1.2. Quality Indicators

To measure the quality of Pareto front, we make use of two indicators in this work: hypervolume [31] and spread [20].

a) ***Hypervolume (HV)***: Hypervolume of the solution set $S = (x_1, \ldots, x_n)$ is the volume of the region that is dominated by S in the objective space. In jMetal, although all objectives are minimized, but the Pareto front is inverted before the hypervolume is calculated. Therefore, the preferred Pareto front would be with the most hypervolume.

b) ***Percentage of Correctness* (%Correct)**: There might be solutions that violate some constraints in the Pareto front, since the correctness is an optimization objective that evolves over time. Solutions that are correct (i.e., without violating any constraint) are more useful to the user; therefore we are interested in the percentage of solutions that are correct in the Pareto front.

Table 2. Feature Models

Repo.	Model	Fea.	Cons.	F_p	F'_p	Ref.
–	JCS	12	13	2	–	–
SPLOT	Web Portal	43	36	4	–	[28]
	E-Shop	290	186	28	–	[29]
LVAT	eCos	1244	3146	54	19	[9, 29]
	uClinux	1850	2468	1244	1244	[30]
	Linux X86	6888	343944	156	94	[9]

5.1.3. Feature Models and Attributes

The details of feature models used in the experiment are summarized in Table 2, with the repository information (*Repo.*), number of features (*Fea.*), number of constraints (*Cons.*), number of prunable features with the preprocessing method in Algorithm 3 (F_p), number of prunable features with the preprocessing method in [13] (F'_p), and literatures (*Ref.*) associated with each feature model.

JCS feature model is the feature model that we have used throughout the paper. Two feature models *Web Portal* and *E-Shop* are from SPLOT respository [17], which is a repository used by many researchers as a benchmark. The *Web Portal* model captures the configurations of Web portal product line, and the *E-Shop* model, which is one of the largest feature models in SPLOT, captures a B2C system with fixed priced products. These two models are chosen to facilitate the comparison with [11]. To further evaluate the scalability of our methods, we make use of feature models from the Linux Variability Analysis Tools (LVAT) feature model repository [18]. The models in LVAT were reversed-engineered by making use of source code, comments and documentations of big projects such as Linux kernel and eCos operating system. Compared to the feature models in SPLOT, the feature models in LVAT contain a significantly larger number of features and constraints, and have higher branching factors, but they have lower ratios of feature groups, and hence shallower tree structures in general.

Table 3. Brief overview of EAs

Algorithm	Population	Operators	Criteria for Domination	Objective of the Criteria
IDEA	Main and Archive for DE; Main and Archive for IBEA	DE mutation, DE crossover, DE selection, Crossover, Mutation, Environmental Selection	The amount of domination are calculated based on quality indicator.	Favors user preferences and absolute domination.
IBEA	Main and Archive	Crossover, Mutation, Environmental Selection	The amount of domination are calculated based on quality indicator.	Favors user preferences.
NSGA-II	Main	Crossover, Mutation, Tournament Selection	Distances to closest point of each objective are calculated. Favors the point with greater distance from other objectives.	Favors more spread out solutions and absolute domination.
ssNSGA-II	Main	Crossover, Mutation, Tournament Selection	Similar to NSGA, with the exception that only one new individual inserted into population at a time.	Favors more spread out solutions and absolute domination.
MOCell	Main and Archive	Crossover, Mutation, Tournament Selection, Random Feedback	Similar to NSGA, a ranking and a crowding distance estimator is used, but bigger distance values are favored.	Favors more spread out solutions and absolute domination.

Note that F_p always contains the same number or more features than F'_p – this shows that our preprocessing method with Algorithm 3 has found more prunable features than [13]. In [13], their preprocessing method is based on static analysis. In particular, they detect always true disjunctions (rules) with only one feature, which means the feature is either a common feature or a dead feature. In addition, they investigate the disjunctions (rules) that include two features, if one of them is prunable in the first round, and the other one could be prunable as well. It is easy to see that our method based on SAT solving could detect all features that could be found by preprocessing method in [13], and it can be shown that F_p is always not fewer than F'_p.

5.1.4. Feature Attribute

Each feature in the feature models has the following attributes, which are the same as the attributes used in [11]:

1. ***Cost*** $\in \mathbb{R}$, records the number of cost incurred to use the feature. For each feature, the *Cost* value is assigned with a real number that is normally distributed between 5.0 and 15.0.
2. ***Used_Before*** $\in \{true, false\}$, indicates whether this feature was used before. The value of *Used_Before* is *true* if the feature has been used before, otherwise it is *false*. For each feature, the *Used_Before* value is assigned with a Boolean value that is distributed uniformly.
3. ***Defects*** $\in \mathbb{Z}$, records the number of defects known in the feature. For each feature, the *Defects* value is assigned with an integer number that is normally distributed between 0 and 10. However, if the feature has not been used before, the *Defects* value is set to 0.

5.1.5. Optimization Objectives

We introduce the five optimization objectives that we use in the experiment in the following. Since jMetal requires minimization of the objectives, all objectives listed here are objectives to be minimized.

Obj1. *Correctness*: minimize the number of violated constraints of the feature model.

Obj2. *Richness of features:* minimize the number of features that are not selected.

Obj3. *Cost:* minimize the total cost.

Obj4. *Feature used before:* minimize the number of features that have not been used before.

Obj5. *Defects:* minimize the number of known defects.

We specify correctness as an objective, rather than a constraint. The reason is that this allows EA to nudge the search towards feature models that contain fewer violated constraints, which eventually leads to valid feature models that do not contain violated constraints. Furthermore, some objectives are conflicting, e.g., $Obj2$ and $Obj3$, because the richness of features would imply a higher cost, but at the same time the cost needs to be minimized.

5.1.6. Configurations of EAs

Given an EA (including our IDEA), we introduce the configurations for comparison.

1. **F+P**: This is the EA that makes use of feedback-directed crossover and mutation (Section 4.2) and preprocessing (Section 4.1) is applied before the execution of the feedback-directed EA.
2. **U+P**: The unguided version of EA with preprocessing (Section 4.1) applied before the execution of the unguided EA. We have demonstrated that, our method has found more prunable features than the preprocessing method of [13] in Section 5.1.3; therefore, U+P can be seen as an improved version of [13] with smaller search space.
3. **U**: The unguided version of EA without preprocessing, which is used by [11, 12].

5.1.7. Parameter Settings

For U, the same as [12], single-point crossover and bit-flip mutation are used as crossover and mutation operators, with crossover and mutation probabilities set to 0.1 and 0.01 respectively. These operators and probabilities also apply to $U + P$. For $F + P$, the feedback-directed crossover (Algorithm 6) and feedback-directed mutation (Algorithm 4) operators are used. The error mutation probability P_{emut}, mutation probability P_{mut}, and crossover probability P_{cross} are set to 1.0, 0.0000001, and 0.1 respectively. Note in Table 6, configuration $F' + P$ is the same as $F + P$, with the exception that the mutation probability P_{mut} is set to 0.01. All other parameter settings for each EA are default settings of jMetal (e.g., population size is set to 100), and therefore are omitted here.

For SPLOT case study, we make use of 25000 evaluations using five EAs (IDEA, IBEA, NSGAII, ssNSGAII, and MoCell). For the larger LVAT case study, we make use of 100000 evaluations using IDEA and IBEA. For both case studies, we generate 10 sets of attributes. For each set of attributes, we run each EA repeatedly for 30 times, and report the medium values of the metrics. The evaluation results for SPLOT and LVAT are reported in Table 4 and Table 6, respectively.

We make use of Mann Whitney U-test [32] to test the statistical significant of *%Correct* indicator. We highlight the *%Correct* in **bold** for $F + P$, if the confidence level exceeds 95% when comparing $F + P$ and $U + P$.

The experiments were conducted on an Intel Core I7 4600U CPU with 8 GB RAM, running on Windows 7.

5.2. Evaluation with SPLOT

Table 4 demonstrates the results with SPLOT case study, where *Time*(*s*), *HV*, and *%Correct* represent execution time in seconds, hypervolume and percentage of correct solutions in the Pareto front. Table 8 also shows the number of non-dominated solutions found by IDEA and IBEA for two sets of feature attribute values.

5.2.1. Answer to RQ1

We notice that the IDEA and IBEA have significantly outperformed other methods on the percentage of correctness, regardless of optimization or parameter settings. This is conformed to the observation in [11]. According to [11], the reason is that all EAs used in this case study (other than IBEA) use diversity-based selection criteria, which favor higher distances between solutions. Thus, non-indicator based methods tend to remove solutions crowded towards the zero-violation point, achieving lower scores on the percentage of correctness measure.

IDEA is actually a little special, as it is a combination of indicator-based method and diversity-based selection criteria. As can be seen from columns 5 and 8 in Table 4, IDEA U outperforms IBEA U version in terms of correctness rate on the three models in SPLOT. For the $U + P$ configuration, IDEA is also much better than IBEA in terms of correctness rate, especially for the two large models *Web Portal* and *E-Shop*.

However, if we consider both preprocessing and the feedback directed mechanisms, IBEA $F + P$ achieves the best correctness rate among all the configuration of all EAs. The correctness rate of IDEA $F + P$ is slightly less than that of IBEA $F + P$ (less than 5% on average), and its HV is correspondingly less than that of IBEA $F + P$ (less than 0.01 on average).

To sum up, without the optimization, IDEA U is significantly better than IBEA U in terms of correctness and HV. When optimization is applied, IDEA $F + P$ is almost close to IBEA $F + P$ in terms of the correctness rate and HV. We attribute the high correctness rate of IDEA to the design of adopting one IBEA population of 50 instances. Despite the slight compromise in correctness made by IDEA, the benefit in attaining more non-dominated solutions is proved to be worthy of such compromise (see Section 5.2.3).

5.2.2. Answer to RQ2

Among all the EAs, although NSGAII and MOCell are extremely fast (less than 30% of runtime of the IBEA), they produce too many invalid solutions, resulting in low correctness ratio. We mainly consider the runtime of IBEA as the benchmark to test the performance of IDEA on SPLOT.

$U + P$ and U configurations. For these unguided configurations U and $U + P$, IDEA takes a bit more runtime than the corresponding configuration of IBEA. This reason is that IDEA needs to perform two environmental

Table 4. Evaluation of IDEA and F+P mechanism with SPLOT

Model		IDEA			IBEA			NSGAII			ssNSGAII			MOCell		
		F+P	**U+P**	**U**	**F+P**	**U+P**	**U**	**F+P**	**U+P**	**U**	**F+P**	**U+P**	**U**	**F+P**	**U+P**	**U**
E-Shop	Time(s)	6.7	7	7.5	7	6.4	7.4	1.9	2.2	2.5	15.2	16.9	17.5	2.8	4	4.5
	HV	0.3	0.28	0.21	0.3	0.18	0.19	0.26	0.2	0.17	0.24	0.22	0.22	0.24	0.19	0.22
	%Correct	**99**	87	11	**100**	0	0	**12**	0	0	**15**	0	0	**14**	0	0
Web Portal	Time(s)	5.6	5.7	6	5.7	4.6	4.6	0.4	0.5	0.5	8.3	8.3	8.2	1	1.8	1.9
	HV	0.31	0.31	0.27	0.32	0.2	0.23	0.3	0.24	0.21	0.3	0.21	0.24	0.31	0.21	0.22
	%Correct	97	98	82	**100**	1	0	**28**	0	0	**20**	1	0	**41**	0	0
JCS	Time(s)	5.1	5.2	5.3	4.7	4.3	4.7	0.3	0.3	0.3	7.3	6.8	6.9	0.3	0.4	0.6
	HV	0.28	0.28	0.24	0.29	0.3	0.28	0.3	0.29	0.28	0.29	0.29	0.29	0.33	0.31	0.3
	%Correct	85	85	64	**86**	78	54	**27**	22	16	**31**	24	14	**34**	21	18

selection processes, which require iteratively calculating the fitness values (i.e., the indicators calculated in Eq.(3)).

$F + P$ **configuration.** On the small-size model *JCS*, IDEA is less efficient than IBEA, taking 8% more time. On the medium-size model *Web Portal*, these two EAs are quite close in performance. We attribute this to the reason of the two environmental selection processes. On the large-size model *E-Shop*, IDEA $F + P$ takes less time (5% less) than IBEA $F + P$. The rationale is that IDEA adopts the DE algorithm that performs well with a small population size on the problem with a large solution space — *E-Shop* has the largest solution space, on which DE may converge faster than IBEA. Thus, we find that IDEA is more stable and more tolerant to the impact of feature model size, showing the similar runtime for different SPLOT models. Thus, IDEA may be less efficient than IBEA due to the two extra selection processes, but it may converge faster in some cases.

5.2.3. Answer to RQ3

We randomly generate two sets of feature attribute values for each model. For the same set of feature attribute values, we apply both the configuration $F + P$ of IDEA and IBEA, as we want to compare our IDEA with the best configuration of IBEA.

For each set of feature attribute values, we execute 30 times and record the results in Table 8. For simplicity, we denote the results of IDEA $F+P$ as **A** and results of IBEA $F + P$ as **B**. We also record the *mean* correctness rate, time and hypervolume of 30 executions of IDEA $F + P$ in column **A %Correct**, **A Time(s)** and **A HV**.

To evaluate the searching capability of IDEA and IBEA in finding non-dominated solutions, we do not compare them based on any single execution that might be biased, instead, based on the union of found non-dominated solutions of 30 executions. We record the *correct* solutions found by IDEA in column $|A|$, and those by IBEA in column $|B|$. Column $|A \cap B|$ shows the number of correct solutions commonly found by IDEA and IBEA, while column $|N_A \cup N_B|$ lists the number of non-dominated solution existing in all solutions found by both EAs out of 30 executions. For ease of comparison, we also list the number of non-dominated solutions found by IDEA in 30 executions in column $|N_A|$; the counterpart found by IBEA is in column $|N_B|$.

Results on *JCS*. IDEA is more suitable for the problem with large search space. On the small model, IDEA and IBEA show the similarity capability in searching for non-dominated solutions. As model *JCS* has a small size of features, the corresponding solution space is also small — not many correct solutions can be found. For the first set of feature attribute values, only 19 and 21 correct solutions are found by IDEA and IBEA, respectively. The solutions found by IDEA are all found by IBEA, as $|A \cap B| = 19$. Due to $|N_A \cup N_B| = 21$, all these solutions are non-dominated ones. Thus, for the first value set, IBEA finds two more non-dominated ones. For the second value set, although IBEA finds 4 more correct solutions than IDEA (12 solutions), these 4 solutions are dominated by others. Hence, for the second value set, IDEA finds the same non-dominated solutions as IBEA.

Results on *Web Portal* and *E-Shop*. As the size of feature model increases, IDEA can find more non-dominated solutions. As the solution space *Web Portal* is not as small as that of *JCS*, IDEA and IBEA share less commonly found solutions (see $|A \cap B|$). Although IBEA finds more correct solutions (508 and 414 for two sets of feature attribute values), most of them are dominated by other solutions, especially those that IDEA finds.

On *E-Shop*, since the feature size is 290, the solution space can be very big. Two EAs share no common correct solutions, and they all report a similar number of correct solutions (1355–1491) after 30 executions. However, it is surprising to see that most of solutions (about 90%) found by IBEA are dominated by those that IDEA finds.

Table 5. Improvement of F+P over U+P for EAs on SPLOT

	-ΔTime (s)	+ΔHV	+Δ%Correct
IDEA	0.2	0.01	4%
IBEA	-0.7	0.08	69%
NSGA-II	0.1	0.05	15%
ssNSGA-II	0.4	0.04	14%
MOCell	0.7	0.06	23%

These results indicate that the DE operators can work more effectively than the traditional GA operators at finding high-quality non-dominated solutions on the *E-Shop* case.

5.2.4. Answer to RQ4

Optimization and correctness rate. We also notice that for each EA, the configuration $U+P$ outperforms the configuration U on the percentage of correctness. This is because the preprocessing method has filtered away the prunable features, which makes the search space smaller. Hence EAs are more effective in the optimal feature selection. We also observe that $F + P$ outperforms $U + P$ constantly on the percentage of correctness. This is attributed to the feedback-directed crossover and mutation, which have effectively guided EAs to explore more promising region of the solution space for locating the optimal feature selection. The average improvement for the configuration $F + P$ over $U + P$ is summarized in Table 5, where the values are calculated by summing up the differences of *%Correct* between $F + P$ and U for all tested EAs, and divided by three (the number of test cases). Positive values mean improvements, while negative values mean the opposite. This has shown that our methods have provided an improvement on the percentage of correctness for all case studies using different EAs, especially in IBEA which has 69% improvement of correctness.

Optimization and runtime. The runtime of configurations $F + P$, $U + P$, and U are comparable. There does not exist a configuration that has a clear advantage over the others in terms of the runtime. The reason is that all configurations go through the same number of evaluations. One might think that the configuration $F + P$ requires an extra calculation of the error position using Algorithm 5. In fact, the constraints also need to be enumerated for configurations $U + P$ and U during each round of evolution, in order to calculate the number of violated constraints. Therefore, the extra operation of $F + P$ is only the *getFeatures* function that is used in line 12 of Algorithm 5, which has a low complexity. On the other hand, $F + P$ and $U + P$ have shorter chromosome than U due to the preprocessing. However, these do not reflect much on the results, because the selection, mutation, and crossover for chromosomes could be done efficiently.

Table 6. Evaluation IDEA and F+P mechanism with LVAT

Model		**IDEA**			**IBEA**		
		F+P	F'+P	U+P	F+P	F'+P	U+P
eCos	Time (s)	47.5	58.2	58.1	35.6	51.3	58.6
	HV	0.24	0.22	0.21	0.24	0.21	0.18
	%Correct	**89**	17.9	6.0	**91**	61	0.0
uClinux	Time (s)	45	44.8	41.9	50.7	43.9	47
	HV	0.3	0.28	0.29	0.31	0.29	0.28
	%Correct	100	98	100	100	100	0.0
Linux X86	Time (s)	2713	2808	15457[1]	2613	3277	12804[2]
	HV	0.21	0.21	0.2	0.2	0.22	0.18
	%Correct	0.0	0.0	0.0	0.0	0.0	0.0

Table 7. Improvement of F+P over U+P for EAs on LVAT

	-ΔTime (s)	+ΔHV	+Δ%Correct
IDEA	3.8[3]	0.02	28%
IBEA	10.9[4]	0.03	64%

5.2.5. Answer to RQ5

We notice that the percentage of correctness of all tested EAs (IDEA, IBEA, NSGA-II, ssNSGA-II and MOCell) have been improved by using $F + P$. These results convey to us that, the preprocessing method and feedback-directed crossover and mutation have provided an advantage on the percentage of correctness and HV, regardless of the underlying EAs. The reason is that the preprocessing method effectively prunes the search space, and the feedback-directed crossover and mutation allow underlying EAs to use the feedback for faster finding of valid solutions. This shows that the preprocessing method and feedback-directed crossover and mutation are general methods that could be applied to different EAs.

5.2.6. Answer to RQ6

To evaluate the scalability of IDEA together with the optimization, we make use of the *E-Shop* model. *E-Shop* model contains one of the largest set of features in the SPLOT repository [17]. The results show that, with $U+P$ and U, except IDEA, none of the EAs could locate

[1] For Linux X86, the jMetal configuration of IDEA $U + P$ that uses cached results throws runtime exceptions of memory limit with 4G memory for JVM. We have to disable cache for IDEA $U + P$, and it greatly increases the time.

[2] Due to the memory limit exception, we have to disable cache to avoid the exception for IBEA $U + P$.

[3] The time measurement is based on using cache results. We ignore the Linux X86 case, as IDEA $U + P$ causes runtime exceptions of memory limit.

[4] We ignore the Linux X86 case, as IBEA $U + P$ using cache throws runtime exceptions of memory limit.

Table 8. Non-dominated solutions found by IDEA F+P (A) and IBEA F+P (B) on SPLOT and LVAT

Model	A %Correct	B %Correct	A Time(s)	B Time(s)	A HV	B HV	$\|A\|$	$\|B\|$	$\|A \cap B\|$	$\|N_A \cup N_B\|$	$\|N_A\|$	$\|N_B\|$
JCS 1	92.4	92.9	5.0	3.5	0.30	0.30	19	21	19	21	19	21
JCS 2	66.9	68.4	5.8	4.3	0.23	0.23	12	16	12	12	12	12
Web Portal 1	99.2	100.0	5.7	5.4	0.31	0.29	272	508	20	213	141	77
Web Portal 2	99.5	100.0	5.6	5.8	0.28	0.28	273	414	23	262	148	129
E-shop 1	99.1	100.0	6.8	7.8	0.29	0.29	1491	1355	0	873	713	160
E-shop 2	98.7	99.9	6.7	9.1	0.30	0.30	1441	1378	0	930	829	101
eCos 1	87.6	92.3	48.2	33.9	0.25	0.25	1306	1533	0	1514	742	772
eCos 2	88.8	92.6	47.9	32.8	0.25	0.25	1337	1621	0	1520	669	851
ucLinux 1	100.0	100.0	44.9	54.8	0.30	0.30	1929	1511	0	968	894	74
ucLinux 2	100.0	100.0	45.6	53.7	0.30	0.30	1875	1620	0	1074	875	199

Table 9. Non-dominated solutions found by IDEA F+P (A) and IBEA F+P (B) that both use 3 common seeds on Linux X86

Model	A %Correct	B %Correct	A Time(s)	B Time(s)	A HV	B HV	$\|A\|$	$\|B\|$	$\|A \cap B\|$	$\|N_A \cup N_B\|$	$\|N_A\|$	$\|N_B\|$
Linux X86 1	28.6	30.2	2495.4	2311.4	0.23	0.23	464	251	0	632	421	213
Linux X86 2	20.3	25.4	2611.3	2549.4	0.23	0.23	344	164	0	447	315	134

a correct solution. On the other hand, with $F + P$, IBEA and IDEA have achieved 100% and 99% of correctness respectively, while NSGA-II, ssNSGA-II and MOCell have achieved 12–14% of correctness. We have also further evaluated for *50 millions* rounds of evolution for $U + P$ for IBEA. It has only achieved *46%* of correctness after 50 millions rounds which takes *3.25 hours*. In contrast, the configuration IBEA $F + P$ has achieved *100%* of correctness by just *7 seconds*. Our IDEA $F + P$ takes even less time only 6.7 seconds to get *99%* of correctness, and IDEA $U + P$ takes *7 seconds* to get *87%* of correctness. After we relax the rounds of evolutions to 50,000, IDEA $U + P$ will reach above *99%* of correctness.

5.3. Evaluation with LVAT

To confirm the scalability of the IDEA, we conduct the evaluation using LVAT. We just show the results of IDEA and IBEA on LVAT models in Table 6, as other EAs cannot find any correct solution on industrial models that are much larger than *E-Shop*. We answer RQ1 to RQ3 in Section 5.3.1 and RQ4 to RQ5 in Section 5.3.2. The successful application of IDEA together with the optimization on LVAT exhibits the scalability of our approach, which answers RQ6. However, EAs on their own fail to find any correct solution on *Linux X86*, we propose to use the seed method for IDEA and IBEA in Section 5.3.3.

5.3.1. Evaluating IDEA

Correctness and performance of IDEA. As can be observed in Table 6, for the configuration $F + P$, IDEA performs quite similarly with IBEA in terms of correctness rate, runtime, and HV. For the configuration $F' + P$, the same situation is observed, except on *eCos* — the correctness of IDEA $F' + P$ (17.9%) is much less than that of IBEA $F' + P$ (61%), as IDEA may mutate too greatly to keep searching near the correct solutions. However, an interesting finding is that the diversity of IDEA helps to work better than IBEA, when the feedback-directed mechanism is disable. Thus, for the configuration of $U + P$, IDEA shows better correctness rate, especially on *uClinux*, IDEA $U + P$ achieves *100%* correctness. The reason is that there are 1244 prunable features in *uClinux* and there are many correct solutions in the solution spaces. Hence, some extent of diversity (a certain probability of mutation) is preferred, especially when the feedback directed mechanism is disabled and no correct solutions have been found yet.

Non-dominated solutions found by IDEA. Neither IDEA nor IBEA can find correct solutions on *Linux X86*. In Table 8, we just show the results for two sets of feature attribute values on *eCos* and *uClinux*. On the former model, IBEA $F + P$ find more correct and also non-dominated solutions for both two value sets. The reason is that *eCos* has very few prunable features, 54 out of 1244 (see Table 2), but as many as 3146 constrains. Differential evolution in IDEA does not help much, as the solution space is small and more mutations may reduce the correctness of the results. In contrast, *uClinux* has more prunable features but less constrains. The solution space is large enough to tolerate some extent of mutations without significantly reducing the correctness. Hence, IDEA attains more correct solutions and also most of the non-dominated ones in $N_A \cup N_B$.

5.3.2. Evaluating Optimization

Table 6 also demonstrates how the optimization improves IDEA and IBEA on LVAT models. Configuration $F' + P$ is the same as $F + P$, with the exception that the mutation probability P_{mut} is set to 0.01 (for $F + P$, $P_{mut} = 0.0000001$). The average improvement for the configuration $F + P$ over $U + P$ is summarized in Table 7.

We notice that for *eCos* and *uClinux*, $F + P$ achieves 90% above correctness for all cases, while $U + P$ does not find any correct solution after 100000 executions. Although $F + P$ achieves overall better runtime, it does not has clear advantage over $U + P$ for all models. These results have confirmed for the better percentage of correctness and comparable runtime of $F + P$ over $U + P$. For *Linux X86* which contains 6888 features, none of the configuration ($F + P$, $U + P$, and U) for each EA has found a correct solution. Therefore, we resort to the "seeding method" proposed by [13].

5.3.3. Seeding Method for Linux X86

In [13], the authors make use of two methods, i.e., SMT solver and IBEA of two objectives, for finding a correct solution (the "seed"), and plant the seed in the initial population of IBEA with the hope to find more valid solutions. In this paper, we also use IDEA of two objectives to generate the seed solution. We run these three seeds (i.e., seed from SMT, IBEA of two objectives and IDEA of two objectives) for both IDEA and IBEA with $F + P$, and compare the results with the seeding method with IBEA proposed by [13], i.e., $U + P$.

Seed 1: from solver. First, Microsoft Z3 SMT solver [33] is used to find a seed. In our case, Z3 successfully finds a valid solution in around three seconds (we repeat for 30 times, and medium of the number of selected features is 1455). With the seed, IBEA $F + P$ successfully find 34 correct solutions using no more than 30 seconds, but IBEA $U + P$ finds no new solution after 30 minutes.

Seed 2: from IBEA. Second, IBEA with two objectives is used to generate seed 2. Using seed 2 to get solutions for 5 objectives, IBEA $F + P$ takes around *40 seconds* to get more than 30 correct solutions. While for IBEA $U + P$, it spends a total of *3.5 hours* of execution time for 30 correct solutions and *4 hours* of execution time for 36 correct solutions. $F + P$ has shortened the search time of $U + P$ for more than 200 times. In particular, $U + P$ spends 3 hours to generate the seed, and spends half an hour to obtain 30 correct solutions. Even given another half hour, $U + P$ finally obtains 36 correct solutions.

Seed 3: from IDEA. We adopt the same way of using two objectives in IBEA to generate seed 3 in IDEA. With seed 3 for all the 5 objective, IDEA $F + P$ takes around *50 seconds* to get more than 20 correct solutions. For IDEA $U + P$, in medium case, it spends around *4 hours* of execution time for getting more than 20 correct solutions, which is a little longer than that in IBEA $F + P$. We also observe that in seed generation, the solutions found by IDEA exhibit more variability due to the diversity based selection criteria.

The quality of three seeds. We observe that the seeds generated by IDEA and IBEA with two objectives, are better than the seed generated by the Z3 SMT solver. These results have shown that $F + P$ outperformed $U + P$ given the two seeding methods in [13] and the seeding method of IDEA with two objectives. Both configurations $F + P$ and $U + P$ of IDEA or IBEA find more solutions using seed 2 or seed 3, compared with seed 1. For example, using seed 2 for IBEA $F + P$ or seed 3 for IDEA $F + P$, they can find around 30 correct solutions on average in 30 executions, while for seed 1 either IDEA $F + P$ or IBEA $F + P$ can find only less than 20 solutions on average. This is conformed to the observation in [13]. According to [13], it is because the seed generated by IBEA with two objectives has more selected features, and the "feature-rich" seed allows the effective search of other valid solutions.

Evaluating with 3 common seeds. To avoid the bias due to different seeds, we include all the three seeds in the initial populations. We run IBEA $F + P$ and IDEA $F + P$ 30 times with random values for feature attributes, and use the medium value for comparison. IDEA gets 20% as correctness, taking 2566 seconds (for 100,000 generations) and achieving HV of 0.23. In contrast, IBEA gets 27% as correctness, taking 2452 seconds and achieving the same HV of 0.23. Although IBEA exhibits a slightly higher correctness, it does not mean IBEA can find more unique and non-dominated solutions. Thus, we randomly pick up two sets of feature attribute values, and compare the specific solutions found by the two algorithms.

In Table 9, we show the non-dominated solutions found by IDEA $F + P$ and IBEA $F + P$ using the three common seeds. Given two sets of random values for feature attributes, IDEA has found 97.7% more and 135.1% more unique and non-dominated solutions on feature attribute set 1 and 2, respectively. Hence, IBEA has a slightly higher correctness, but find less unique correct solutions — $|B|$ is much smaller than $|A|$ for both value sets of feature attribute. The reason is that the solutions found by IBEA according to the three seeds are quite repetitive in 30 executions. Thus, for the same seeds, solutions found by IBEA are less diverse, compared with IDEA. The lower diversity of IBEA solutions also leads to the less number of non-dominated solutions, com-

pared with IDEA, for both sets of feature attribute values.

We also compare how the mutation parameter P_{mut} affects feedback-directed IBEA. Out of five models, $F' + P$ only performs poorer than $F + P$ in *eCos*. To better observe the effect, we make use of *E50*. It shows that $F + P$ obtained 50% of correct solutions in Pareto front in a smaller number of evaluations for all models, except *LinuxX86*. The results show that smaller P_{mut} leads to faster convergence of correct solutions in Pareto front. This is because smaller P_{mut} minimizes the modification of non-error positions; therefore, it allows IBEA to focus more on the correction of constraint violations.

5.4. Threats to Validity

There are several threats to validity. The first threat of validity is due to the fact that values for the feature attributes (i.e., *Cost*, *Defects*, and *Used_Before*) were randomly generated. This is due to difficulty in obtaining the attributes that are associated with real-world products since many of them are proprietary. To mitigate the effect of randomness, we generate 10 set of attributes for each case study. Furthermore, for each set of attributes, we run each EA repeatedly for 30 times, and report the medium values of the metrics. Future work should involve the use of real data for the evaluation.

The second threat of validity stems from our choice of using an exemplar parameter set (e.g., for crossover and mutation probability), which comes with the default setting of jMetal, in order to cope with the combinatorial explosion of options. To address these threats, it is clear that more experimentations with different feature models and experimental parameters are required, so that we could investigate effects that have not been made explicit by our dataset and experimental parameters.

6. Related Work

Our work is related to the feasible feature selection. In [34], White *et al.* reduced the feature selection problem in SPL to a multidimensional multi-choice knapsack problem (MMKP). They proposed a polynomial time approximation algorithm, called Filtered Cartesian Flattening (FCF), to derive an optimal feature configuration subject to resource constraints. Their evaluation showed that FCF can stably achieve the optimality above 90% even when the number of resources increases up to 91, while the optimality of Constraint Satisfaction Problem (CSP) based Feature Selection in [35] drops down to 30% when there are 91 resources.

Although FCF in [34] can achieve a highly optimal solution, but it requires significant computing time. To address the problem of scalability, Guo *et al.* [10] presented GAFES (a genetic algorithm based approach). The rationale is that GAs are quite suitable for the highly constrained problems, such as the feature selection (product derivation) problem. GAFES integrated a new *repair* operator for feature selection and also defined a *penalty* function for resource constraints. The evaluation showed GAFES may not beat the FCF and CSP in optimality, but it scaled up to large-scale models with a reasonable optimality.

Genetic algorithm only allows single objective function, and in addition, the method proposed in [10] repairs each solution explicitly, and does not take advantage of the evolution of GA algorithm for repairing. To address this problem, Sayyad *et al.* [11] investigated the use of different types of EAs that support multi-objective function for the optimal feature selection. They adopted 7 types of EAs, such as IBEA, NSGA-II and MOCell, to search for the optimal product. The results have shown that IBEA performs much better than other 6 EAs in terms of time, correctness and satisfaction to user preferences. In [12], Sayyad *et al.* improved [11] by turning down the crossover probability from 0.9 to 0.1 and mutation probability from 0.05 to 0.01, and they reported HV-mean and spread mean may increase by 5% to 10% in most cases. In [13], Sayyad *et al.* proposed the use of EA with simple heuristic in larger product lines from LVAT repository. They proposed the use of static analysis to identify prunable features for reducing search space, and the use of seeded techniques to find more correct products from Linux X86 Kernel. Our method has improved the method proposed in [11, 12, 13] by incorporating feedback-directed mechanism for EAs (cf. Section 5 for the evaluation). We also show that our method for finding prunable feature with Algorithm 3 is always not fewer than the method proposed in [13] (cf. Section 5.1.3 for the explanation and evaluation).

Our method is relevant to searching valid features for a feature model. In [36], an experiment for measuring the efficiency of BDD, SAT, and CSP solvers is conducted using feature models from SPLOT repository. They reported long run times for certain operations, and certain runs are cancelled if exceeded three hours. They also reported an exponential runtime increase with the number of features for non-BDD solvers on the “valid" operations. In [37], the state-of-art solvers, e.g., JavaBDD BDD solver, the JaCoP CSP solver and the SAT4J SAT solver, were used to answer the questions such as “derive one valid product from a feature model” and “number of products”. They found that CSP and SAT solvers have exponential runtime increase as the feature size of feature model increases, and BDD requires a maximum of 28 seconds

to derive a valid product for web-portal, even without considering the quality of feature attributes. Thus, these automated reasoning techniques can be precise, but generally not scalable for large feature models. Our work complements with their work by using feedback-directed evolutionary algorithm that scales well for large feature models. In [38] introduces five novel parallel algorithm for Multi-Objective Combinatorial Optimization (MOCO) to allow parallel processing. Our work complements with them by considering feedback-directed mechanism for MOCO problem using feedback-directed EAs.

Our work is also related to the feedback-directed methods in software engineering. Pacheco *et al.* [39] proposed RANDOOP, a feedback-directed mechanism for performing random test. It uses erroneous results of previous method invocation to generate a better random test. Clarke *et al.* [40] proposed CEGAR, which uses spurious counterexamples as a feedback to guide the refinement process. Our method is on feedback-directed methods in EAs for the optimal feature selection. This work is also related to using evolutionary algorithms and SMT solvers in tackling software engineering problems. In [41], we make use of genetic algorithm in calculating the optimal recovery plan during service failure. In [42, 43], we calculate the local time requirements of individual components given the global time requirement of the system with Z3 SMT solver [33]. In this work, our focus in on making use of evolutionary algorithms and Z3 SMT solver in tackling optimal feature selection.

In addition to the SPL domain, multi-objective evolutionary optimization algorithms (MEOAs) have also been applied to various software engineering problems. In [44], Harman *et al.* proposed the term Search-Based Software Engineering (SBSE), and reported that the surveyed and proposed optimization techniques for SE problems by 2001 were all single-objective based. Seeing the potential of using multi-objective optimization, Harman [45] discussed about the possible usage of the meta-heuristic search techniques such as: simulated annealing and genetic algorithm. Harman considered it insensible combination of multiple metrics into an aggregate fitness in the way of assigning coefficients, and suggested to use Pareto optimality rather than aggregate fitness.

7. Conclusion and Future Work

In this study, we have presented a novel approach — Indicator-based Differential Evolutionary Algorithm (IDEA) to achieve both correctness and diversity of the found results. IDEA maintains two populations, one IBEA population for correctness and one DE population for diversity of results. Besides, we combine two optimization techniques, i.e., feedback-directed mechanism and preprocessing, into various EAs. Based on analyzing violated constraints, we use the analyzed results as a feedback to guide the process of crossover and mutation operators. In addition, we also introduce the preprocessing to reduce the search space, by filtering away the prunable features in all feasible feature sets. Our evaluation shows that except the case of *eCos*, IDEA can find the same (on *JCS*) or more unique and non-dominated solutions (on others) than IBEA. Generally, IDEA sacrifices a little in correctness and runtime, but find more non-dominated solutions in return.

Furthermore, the preprocessing technique and the feedback-directed mechanism have both improved over existing unguided EAs on the optimal feature selection. The feedback-directed IBEA successfully found 69% and 64% more correct solutions for case studies in S-PLOT and LVAT repositories, compared to the unguided IBEA. Due to its own diversity of IDEA, the feedback mechanism only improves the correctness by 4% and 28%. In addition, with “seeding method" proposed by [13] and feedback-directed IBEA or IDEA, we have reduced the running time from about *3.5 – 4 hours* to about *40–50 seconds* to find more than 30 correct solutions on *Linux X86*.

As future works, first, we plan to find other types of feedback that could be incorporated in EAs, to address the scalability problem of large feature models, such as Linux X86. Second, we would investigate extensibility of our method to other software engineering problems. Lastly, we plan to further evaluate the method using different case studies.

References

[1] K. C. Kang, J. Lee, P. Donohoe, Feature-oriented project line engineering, IEEE Software 19 (4) (2002) 58–65.

[2] P. Clements, L. Northrop, L. M. Northrop, Software Product Lines : Practices and Patterns, Addison-Wesley Professional, 2001.

[3] L. Chen, M. A. Babar, N. Ali, Variability management in software product lines: a systematic review, in: Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings, 2009, pp. 81–90.

[4] M. Sinnema, S. Deelstra, Classifying variability modeling techniques, Information & Software Technology 49 (7) (2007) 717–739.

[5] I. Jacobson, M. L. Griss, P. Jonsson, Software reuse - architecture, process and organization for business, Addison-Wesley-Longman, 1997.

[6] J. Karlsson, S. Olsson, K. Ryan, Improving practical support for large-scale requirement prioritising, Requir. Eng. 2 (1) (1997) 51–60.

[7] M. L. Griss, J. Favaro, M. d. Alessandro, Integrating feature modeling with the rseb, in: Proceedings of the 5th International Conference on Software Reuse, ICSR '98, 1998, pp. 76–.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented domain analysis (foda) feasibility study, Tech. Rep. CMU/SEI-90-TR-21, Carnegie Mellon University (November 1990).

[9] S. She, R. Lotufo, T. Berger, A. Wasowski, K. Czarnecki, Reverse engineering feature models, in: ICSE, 2011, pp. 461–470.

[10] J. Guo, J. White, G. Wang, J. Li, Y. Wang, A genetic algorithm for optimized feature selection with resource constraints in software product lines, Journal of Systems and Software 84 (12) (2011) 2208–2221.

[11] A. S. Sayyad, T. Menzies, H. Ammar, On the value of user preferences in search-based software engineering: a case study in software product lines, in: ICSE, 2013, pp. 492–501.

[12] A. S. Sayyad, J. Ingram, T. Menzies, H. Ammar, Optimum feature selection in software product lines: Let your model and valuesguide your search, in: CMSBSE, 2013, pp. 22–27.

[13] A. S. Sayyad, J. Ingram, T. Menzies, H. Ammar, Scalable product line configuration: A straw to break the camel's back., in: ASE, 2013.

[14] E. Zitzler, S. Künzli, Indicator-based selection in multiobjective search, in: PPSN, 2004, pp. 832–842.

[15] T. H. Tan, Y. Xue, M. Chen, J. Sun, Y. Liu, J. S. Dong, Optimizing selection of competing features via feedback-directed evolutionary algorithms, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015, 2015, pp. 246–256.

[16] S. Das, P. N. Suganthan, Differential evolution: a survey of the state-of-the-art, Evolutionary Computation, IEEE Transactions on 15 (1) (2011) 4–31.

[17] M. Mendonça, M. Branco, D. D. Cowan, S.P.L.O.T.: software product lines online tools, in: OOPSLA Companion, 2009, pp. 761–762.

[18] Linux Variability Analysis Tools (LVAT) Repository, https://code.google.com/p/linux-variability-analysis-tools/source/browse/?repo=formulas.

[19] K. Czarnecki, U. W. Eisenecker, Generative programming - methods, tools and applications, Addison-Wesley, 2000.

[20] K. Deb, S. Agrawal, A. Pratap, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-II, IEEE Trans. Evolutionary Computation 6 (2) (2002) 182–197.

[21] J. J. Durillo, A. J. Nebro, F. Luna, E. Alba, On the effect of the steady-state selection scheme in multi-objective genetic algorithms, in: EMO, 2009, pp. 183–197.

[22] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, E. Alba, Mocell: A cellular genetic algorithm for multiobjective optimization, Int. J. Intell. Syst. 24 (7) (2009) 726–746.

[23] J. Zhong, Y.-S. Ong, W. Cai, Self-learning gene expression programming, Evolutionary Computation, IEEE Transactions on PP (99) (2015) 1–1.

[24] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. H. Obbink, K. Pohl, Variability issues in software product lines., in: PFE, 2001.

[25] SAT4J – The boolean satisfaction and optimization library in Java, http://www.sat4j.org/.

[26] M. Srinivas, L. M. Patnaik, Adaptive probabilities of crossover and mutation in genetic algorithms, IEEE Transactions on Systems, Man, and Cybernetics 24 (4) (1994) 656–667.

[27] J. J. Durillo, A. J. Nebro, jmetal: A java framework for multi-objective optimization, Advances in Engineering Software 42 (10) (2011) 760–771.

[28] M. Mendonça, T. T. Bartolomei, D. D. Cowan, Decision-making coordination in collaborative product configuration, in: SAC, 2008, pp. 108–113.

[29] Y. Xue, Z. Xing, S. Jarzabek, Understanding feature evolution in a family of product variants, in: WCRE, 2010, pp. 109–118.

[30] T. Berger, S. She, R. Lotufo, K. Czarnecki, T. Berger, S. She, R. Lotufo, A. Wasowski, K. Czarnecki, Variability modeling in the systems software domain, Tech. rep., University of Waterloo (2012).

[31] E. Zitzler, L. Thiele, Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach, IEEE Trans. Evolutionary Computation 3 (4) (1999) 257–271.

[32] A. Arcuri, L. C. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering., in: ICSE, 2011, pp. 1–10.

[33] L. M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: TACAS, 2008, pp. 337–340.

[34] J. White, B. Dougherty, D. C. Schmidt, Selecting highly optimal architectural feature sets with filtered cartesian flattening, Journal of Systems and Software 82 (8) (2009) 1268–1284.

[35] D. Benavides, P. T. Martín-Arroyo, A. R. Cortés, Automated reasoning on feature models, in: CAiSE, 2005, pp. 491–503.

[36] R. Pohl, K. Lauenroth, K. Pohl, A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models., in: ASE, 2011, pp. 313–322.

[37] R. Pohl, V. Stricker, K. Pohl, Measuring the structural complexity of feature models, in: ASE, 2013, pp. 454–464.

[38] J. Guo, E. Zulkoski, R. Olaechea, D. Rayside, K. Czarnecki, S. Apel, J. M. Atlee, Scaling exact multi-objective combinatorial optimization by parallelization, in: ASE, 2014.

[39] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball, Feedback-directed random test generation, in: ICSE, IEEE Computer Society, 2007, pp. 75–84.

[40] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: CAV, 2000, pp. 154–169.

[41] T. H. Tan, M. Chen, É. André, J. Sun, Y. Liu, J. S. Dong, Automated runtime recovery for qos-based service composition, in: WWW, 2014, pp. 563–574.

[42] T. H. Tan, É. André, J. Sun, Y. Liu, J. S. Dong, M. Chen, Dynamic synthesis of local time requirement for service composition, in: ICSE, 2013, pp. 542–551.

[43] Y. Li, T. H. Tan, M. Chechik, Management of time requirements in component-based systems, in: FM, 2014, pp. 399–415.

[44] M. Harman, B. F. Jones, Search-based software engineering, Information & Software Technology 43 (14) (2001) 833–839.

[45] M. Harman, The current state and future of search based software engineering, in: FOSE, 2007, pp. 342–357.