A genetic algorithm for optimized feature selection with resource constraints in software product lines

Article *in* Journal of Systems and Software · December 2011

DOI: 10.1016/j.jss.2011.06.026 · Source: DBLP

Jianmei Guo
University of Waterloo

Yinglin Wang
Shanghai Jiao Tong University

A Genetic Algorithm for Optimized Feature Selection with Resource Constraints in Software Product Lines

Jianmei Guo[a,*], Jules White[b], Guangxin Wang[a], Jian Li[a], Yinglin Wang[a]

[a]*Department of Computer Science and Engineering, Shanghai Jiao Tong University, 800 Dong Chuan Road, Minhang, Shanghai 200240, China*
[b]*Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA 24060, USA*

Abstract

Software product line (SPL) engineering is a software engineering approach to building configurable software systems. SPLs commonly use a feature model to capture and document the commonalities and variabilities of the underlying software system. A key challenge when using a feature model to derive a new SPL configuration is determining how to find an optimized feature selection that minimizes or maximizes an objective function, such as total cost, subject to resource constraints. To help address the challenges of optimizing feature selection in the face of resource constraints, this paper presents GAFES, an artificial intelligence approach, based on genetic algorithms (GAs), for optimized feature selection in SPLs. Our empirical results show that GAFES can produce solutions with 86-97% of the optimality of other automated feature selection algorithms and in 45-99% less time than existing exact and heuristic feature selection techniques.

Keywords: Software product lines, Product derivation, Feature models, Optimization, Genetic algorithm

1. Introduction

Software product lines (SPLs) are a software engineering approach for creating configurable software applications that can be adapted to a variety of requirement sets (Clements & Northrop, 2001). SPLs are built around a set of common software components with points of variability that allow the customization of the product. For example, the Bold Stroke SPL (Boeing, 2002), developed by the Boeing company, comprises a wide range of reusable artifacts, such as a configurable architecture, set of application components, processes, and development tools, which are used to create Operational Flight Programs for a variety of Boeing aircraft.

Developing an SPL comprises two distinct development processes: 1) *domain engineering*, which is the process of developing the common and variable software artifacts of the SPL (Pohl et al., 2005) and 2) *product derivation*, which is the process of configuring the reusable software artifacts for a customer- or market-specific set of requirements. The core principle behind SPL engineering is that the increased cost of developing a set of reusable and customizable software artifacts can be amortized across multiple products (Deelstra et al., 2005), such as different Boeing aircraft.

An important component of an SPL is a model or roadmap that dictates the rules governing how the points of variability can be configured. A widely used approach for modeling the commonalities and variabilities in an SPL is Feature-Oriented Product Line Engineering, which captures and represents the commonalities and variabilities of systems in terms of features (Kang et al., 1990, 1998, 2002). *Features* are essential abstractions of product characteristics

[*]Corresponding author. Tel.: +86-21-34204415; fax: +86-21-34204728.

Email addresses: `guojianmei@sjtu.edu.cn` (Jianmei Guo), `julesw@vt.edu` (Jules White), `wgxin@sjtu.edu.cn` (Guangxin Wang), `li.jian@sjtu.edu.cn` (Jian Li), `ylwang@sjtu.edu.cn` (Yinglin Wang) relevant to customers and are typically increments of functionality (Kang et al., 2002; Batory et al., 2006).

Every *product* derived from a feature-oriented SPL is represented by a unique and valid combination of features. The valid combinations of features are governed by the SPL's *feature model*, which defines relationships between the features in an SPL. As shown in Figure 1, a feature model is a tree-like structure that defines the relationships among features in a hierarchical manner. The relationships between the features indicate the choices that the customer can make when customizing a product. For example, in Figure 1, the *Alternative* relationship between the two features "Static" and "Dynamic" indicates that developers can either choose a static or dynamic mechanism for implementing the memory allocation (the "MemAlloc" feature) in the database, but not both. The goal of product derivation is to select an optimal combination of features from the feature model that meets the stakeholder requirements.

Open Problem ⇒ SPL Feature Selection Optimization with Resource Constraints. Despite the benefits of SPLs, industrial case studies have shown that product derivation is still "a time-consuming and expensive activity" (Deelstra et al., 2004, 2005). Developers face a number of challenges when attempting to derive an optimized feature selection that meets an arbitrary set of requirements. For example, developers of the database SPL shown in Figure 1 may need to derive the set of features that meets a set of functional requirements for a mobile application while simultaneously minimizing memory consumption (the optimization goal). Some configurations of the database may reduce memory consumption but require more disk space than can be supported by the mobile platform (a resource constraint).

Even in a small feature model, feature combinatorics can produce an exponential number of product configurations. For example, about 280 different product configurations can be derived from the relatively simple feature model shown in Figure 1. Moreover, once a feature selection is made, it must be verified to conform to the myriad constraints in the feature model. Additional resource constraints or non-functional requirements, such as the binary size, performance, or total budget for a product (Siegmund et al., 2008; White et al., 2009), make the feature selection process even more complex and difficult. Previous research has shown that finding an optimal feature selection that conforms to both the feature model constraints and the resource constraints is an NP-hard problem (White et al., 2009). To overcome the complexity of selecting an optimized feature selection that meets resource constraints, developers need tools to help automate the process.

SPL feature selection optimization with resource constraints is similar to configuration optimization problems that have been addressed by other automated feature selection approaches that do not consider resource constraints (Benavides et al., 2008). Many researchers in the SPL community have applied and extended various AI techniques to solve the SPL feature selection problem, e.g., using CSP (Benavides et al., 2005), BDD (Czarnecki & Wasowski, 2007; Mendonca et al., 2009) and SAT solvers (Batory, 2005), but have not considered resource constraints in their work. Moreover, these exact techniques show exponential time complexity and do not scale to large industrial feature models with hundreds or thousands of features (White et al., 2009). Other researchers have developed polynomial-time approximation algorithms for selecting highly optimal feature sets (White et al., 2009), but their approaches still require significant computing time. In addition, some visualization techniques (Sellier & Mannion, 2007; Botterweck et al., 2007) have been devised to assist developers in feature selection. However, industrial-sized feature models can have hundreds or thousands of features (Steger et al., 2004; Loesch & Ploedereder, 2007), which makes a manual feature selection process challenging, particularly for an NP-hard feature selection problem with resource constraints.

Solution Approach ⇒ Genetic Feature Selection Optimization Algorithms. In this paper, we introduce an AI approach, based on genetic algorithms (GAs), to SPL feature selection optimization with resource constraints. A GA is a stochastic and global search heuristic that mimics natural evolution (Mitchell, 1996). It can quickly scan a vast population of possible solutions. GAs often work well for highly constrained problems, such as the Traveling Salesman (Muhlenbein, 1989) and Satisfiability problems (De Jong & Spears, 1989).

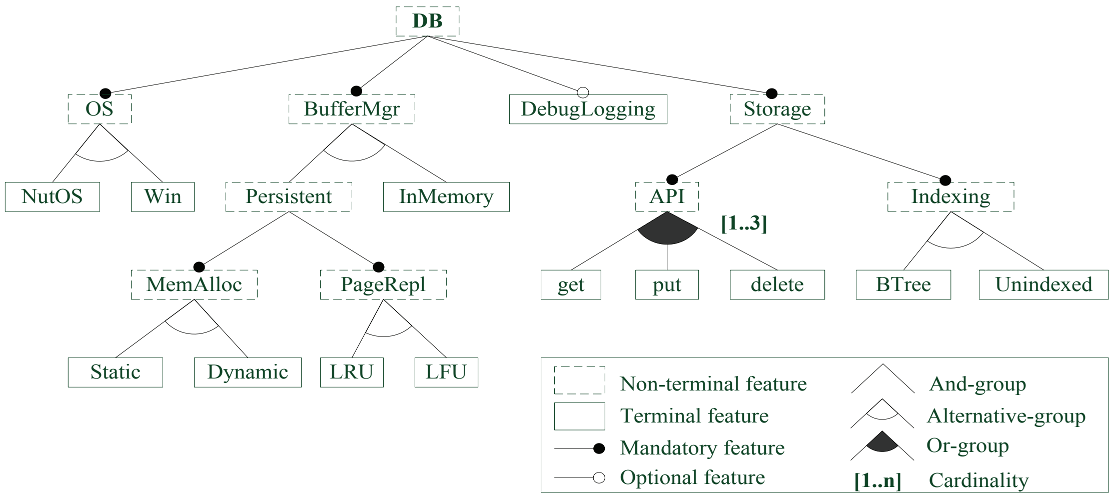

Figure 1: Example feature model for the FAME-DBMS SPL

In this paper, we adapt GAs to the SPL feature selection problem, and propose a GA-based AI approach, called *GAFES*. GAFES can quickly derive an optimized feature selection by evaluating different configurations that both optimize product capabilities and honor resource limitations. GAFES combines a novel feature selection repair operator and a penalty function for resource constraints to obtain fast feature selection times. Our empirical results show that GAFES can produce feature selections with objective function scores that are within 86-97% of the feature selections produced by prior AI feature selection techniques, such as FCF (White et al., 2009) and CSP/SAT-based approaches (Benavides et al., 2005; Batory, 2005; Czarnecki & Wasowski, 2007), for feature models whose size ranges from 10 to 10,000 features. Moreover, in our experiments, GAFES derives feature selections 45-99% faster than these existing heuristic and exact feature selection optimization techniques.

The main contributions of this paper to the study of AI-based optimized feature selection for SPLs are summarized as follows:

- We show how to adapt GAs to the SPL feature selection optimization with resource constraints and propose a modified GA named GAFES to derive an optimized feature selection subject to the feature model constraints and the resource constraints.

- We present a repair operator that allows a GA or other evolutionary algorithm to transform an arbitrary feature set into a valid feature combination that conforms to the feature model constraints.

- We describe a penalty function, based on the ratio of objective function value to consumed resources, that can improve the search process of a GA or evolutionary algorithm to that meets resource constraints.

- We present empirical results from experiments on feature models with 10 to 10000 features. The results show that GAFES can produce feature selections that are within objective function scores that are within 86-97% of feature selections produced by other feature selection techniques. GAFES, however, derives feature selections 45-99% faster than existing heuristic and exact feature selection techniques.

The remainder of this paper is organized as follows: Section 2 contains a brief introduction to feature models; Section 3 presents a motivating example; Section 4 formalizes the SPL feature selection problem; Section 5 discusses the challenges of adapting GAs to the SPL feature selection problem; Section 6 details our genetic feature selection optimization algorithm, called GAFES, and analyzes its algorithmic complexity; Section 7 introduces our experiments with feature model generation and test pattern generation, and presents our empirical results;

Section 8 compares our work with related work; and Section 9 presents concluding remarks and lessons learned.

2. Feature Modeling Background

Figure 1 shows a partial feature model of an embedded database SPL called FAME-DBMS inspired from (Rosenmuller et al., 2008; Thum et al., 2009). A feature model is a tree of features. Every node in the tree has one parent except the *root feature* (e.g., 'DB'). A *terminal* feature (e.g., 'NutOS') is a leaf and a *non-terminal* feature (e.g., 'OS') is an interior node of a feature diagram (Batory, 2005; Thum et al., 2009). Every non-terminal feature represents a composition of features that are its descendants.

A feature model is organized hierarchically and is graphically depicted as an AND-OR *feature diagram* (Kang et al., 1990). *Cross-tree constraints* are used to represent non-hierarchical composition rules comprising mutual dependency (*requires*) and mutual exclusion (*excludes*) relationships (Kang et al., 1990). There are two cross-tree constraints in Figure 1, "LRU requires BTree" and "LFU requires BTree".

New products are derived from a feature model by finding a *configuration* of terminal features (Thum et al., 2009). A feature selection represents a specific product satisfying customer requirements. The actual resource consumption and the benefits of a product can be calculated from the set of terminal features (White et al., 2009). For example, a feature selection from the FAME-DBMS feature model can be used to calculate the amount of memory consumed by the buffer management configuration that is contained in the feature selection.

A feature selection is *valid* if the selection of features is allowed by the constraints described in the feature model. Connections between a feature and its group of children define the constraints on feature selection. The constraints that can be used to govern the allowable child feature selections are *And-* (e.g., 'OS', 'BufferMgr', 'DebugLogging', and 'Storage'), *Or-* (e.g., 'get', 'put' and 'delete'), and *Alternative*-groups (e.g., 'NutOS' and 'Win'). The members of And-groups can be either *mandatory* (e.g. 'OS', 'BufferMgr', and 'Storage') or *optional* (e.g. 'DebugLogging'). Or-groups and Alternative-groups have their own *cardinalities* (Czarnecki & Wasowski, 2007). For example, at least 1 and at most 3 API ('get', 'put' and 'delete') must be selected. Other constraints have been proposed, such as cardinality constraints (Czarnecki & Wasowski, 2007), but we focus on the core constraints that are common across all feature modeling approaches.

The rules for selecting features from a feature model can be summarized as follows (Guo & Wang, 2010): if a feature is selected, its parent must also be selected. If a feature is selected, all of its mandatory children participating in an And-group must be selected. For example, in Figure 1, "Storage" has a mandatory sub-feature "API", which must also be selected if "Storage" is selected. If the selected feature has an Or-group containing children, at least one child must be selected, and in Alternative-groups, exactly one child is selected. For example, at leat 1 and at most 3 child features of "API" can be selected. "Indexing" requires the selection of either of its "BTree" or "Unindexed" sub-features, but not both.

3. Motivating Example

Figure 2 shows a scenario of a sensor network that applies FAME-DBMS, which is excerpted from (FAME-DBMS, 2002). In this motivating example, sensor nodes measure values for temperature, air pressure, and luminance in a biological environment to monitor the growth conditions of different plants. A biological scientist can also manually augment the sensor data by entering additional information about the plants into a PDA. Scientists can use a PDA to retrieve information from the sensor network through a data access point. All the data is stored in a server for further analysis. In every embedded system of this scenario, such as the sensor nodes and the PDA, the FAME-DBMS is used to store and retrieve the data.

The key to product derivation is to select a good feature combination from a feature model according to the requirements of customers and vendors. In practice, feature selection must consider the trade-off between the resource consumption and value of the target product and the limited vendor budget. In the motivating example, the resource consump-

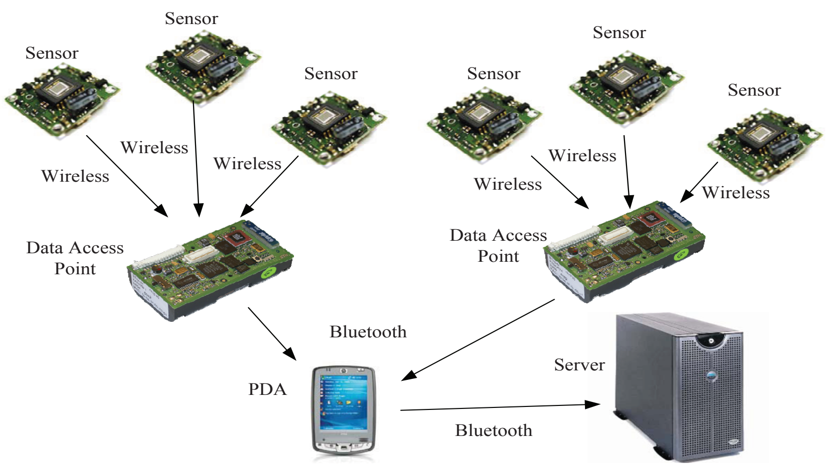

Figure 2: A sample application scenario for the FAME-DBMS

Table 1: Example resource consumption and performance of a subset of the FAME-DBMS features

Feature	Perf.	CPU	Mem.	Cost
DB/OS/NutOS	8	4	2048	10
DB/OS/Win	5	2	1024	15
...				
DB/DebugLogging	2	3	256	50
...				
DB/Storage/Indexing/Btree	8	4	512	30
DB/Storage/Indexing/Unindexed	4	2	128	20

tion comes from the consumption of CPU, memory, available budget, and development staff time (White et al., 2009). Regardless of what configuration of the system is chosen, the resource consumption must not exceed the available values of the target platform. For example, the total memory consumed by the database deployed to the PDA cannot be more than the memory available on the PDA. Moreover, the product may need to be optimized for a specific characteristic, such as minimizing the total cost of each node in the sensor network or maximizing maintainability (Siegmund et al., 2008), performance, stability, etc.

Take the feature model shown in Figure 1 for example, Table 1 shows example information about the provided performance and the consumed resources of a subset of the FAME-DBMS features. Each feature is identified by the path from the root feature in the model to it. In this example, customers want a target product (an embedded database) with maximum performance, but that fits within a limited budget. This budget constraint is a resource limitation, e.g., CPU $\leqslant$ 40, Memory $\leqslant$ 4096, and Cost $\leqslant$ 200. Thus the problem is to derive a target product with the maximum performance subject to the resource limitations. For feature models with hundreds or thousands of features and a large number of feature combinations, choosing an optimized feature selection is hard.

4. Feature Selection with Resource Constraints Problem Formalization

More formally, SPL feature selection optimization with resource constraints can be defined as follows. Let $F = \{f_i\}, 1 \leqslant i \leqslant n$ denote all n features defined in a feature model and C all the dependency constraints and cross-tree constraints depicted by the arcs in the feature diagram, e.g., in Figure 1, 'OS', 'BufferMgr', and 'Storage' are required child features of 'DB'. Every feature $f_i \in F$ has an associated resource consumption, $r(f_i) \in Z$, and a provided value, $v(f_i) \in Z$. $R(F)$ indicates the set of resources consumed by all the features (e.g., columns 3-5 in Table 1) and $V(F)$ the set of values provided by all the features (e.g., columns 2 in Table 1). Rc includes the resource constraints, e.g., CPU $\leqslant$ 40, Memory $\leqslant$ 4096, and Cost $\leqslant$ 200.

Here, we only consider the resource consumed by and the value provided by terminal features because the actual resource consumption and the benefits of a product mainly come from terminal features (White et al., 2009). That is to say, the resource consumed by and the value provided by all the non-terminal features are 0. Thus:

Definition 1. *Given a feature model with n features $F = \{f_i\}, 1 \leqslant i \leqslant n$ and a set of constraints C, the goal of the feature selection problem is to find a feature subset $S \subseteq 2^F$ from all valid feature combinations defined in the feature model such that*

$$V(S)\ (i.e., \sum_{f_i \in S} v(f_i))\ \textit{is maximized} \tag{1}$$

subject to

$$S \longrightarrow_{conforms\ to} C \tag{2}$$

and

$$R(S)\ (i.e. \sum_{f_i \in S} r(f_i))\ \leqslant\ Rc \tag{3}$$

for the resource constraint $Rc \in Z^+$.

5. Challenges of Adapting GAs to SPL Feature Selection Optimization

To make well-informed configuration decisions, developers need the ability to easily generate and evaluate different feature selections that both satisfy resource limitations and optimize specific product capabilities, e.g., minimizing total cost or required CPU and memory in the motivating example. For example, developers of the FAME-DBMS system might need to compare two different SPL configurations that are optimized to minimize memory consumption. In one configuration, the developers might opt to slightly relax their budget constraint to determine what additional performance and storage capacity could be obtained for slightly more money.

Generating valid feature selections and evaluating them is computationally complex and time consuming and thus these types of comparative analysis are difficult. From Definition 1, we can see that the SPL feature selection optimization problem with resource constraints is a highly constrained problem. The SPL feature selection optimization with resource constraints has been proven to be NP-hard and thus exact algorithms for the problem do not scale well (White et al., 2009).

One approach to addressing this problem is to use an AI technique, such as a GA, to automate the feature selection process. However, there are a number of challenges to develop a GA for SPL feature selection optimization with resource constraints. In the remainder of this section, we show that there are three key challenges to developing a GA for optimized feature selection with resource constraints: 1) randomly generating initial solutions tends to produce invalid starting points for a GA; 2) traditional GAs for general feature selection (Yang & Honavar, 1998; Oh et al., 2004) generate an arbitrary feature set, which may not conform to the feature model constraints; and 3) when facing feature selection problems that include resource constraints, GAs require some form of repair or penalty approach , which is hard to derive.

5.1. Challenge 1: Randomly Seeding the Initial Population with Valid Feature Selections

A GA is a stochastic algorithm that mimics natural evolution. Its basic steps and running mechanism are shown in Figure 3. The GA first maintains an initial population of solutions (called individuals or *chromosomes*). A binary encoding is used to represent a potential solution as a chromosome (*e.g.* string, such as “10010101”). As in the case of biological evolution, the evolutionary process begins with an initial population of chromosomes and proceeds over a series of generations. In each generation, every chromosome in the current population is evaluated by a fitness function, which is an objective function that determines the optimality of a chromosome so that chromosomes can be compared against each other. The best chromosomes are selected as parents, and undergo genetic operations, such as crossover and mutation, to generate a new offspring from them. The offspring then replaces a member of the population in the next generation based on a replacement policy. The evolutionary process continues until a chromosome satisfying a minimum criteria is found or a fixed number of generations reaches.

To employ a GA for feature selection, a group of feature sets must be generated randomly to obtain an initial population. However, since they are generated randomly, these initial feature sets may not be valid feature combinations that conform to the fea-

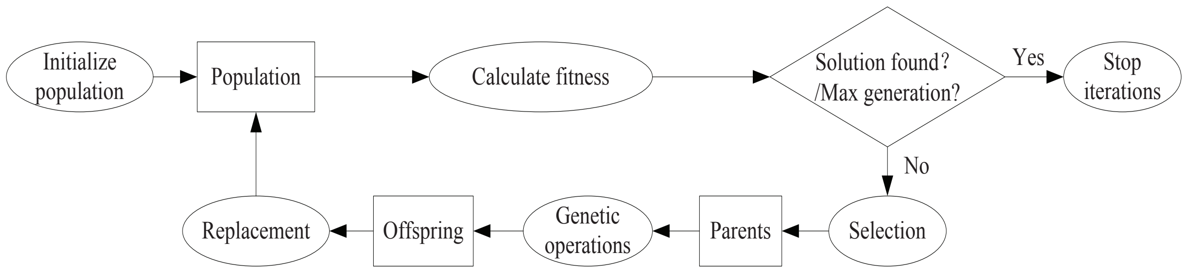

Figure 3: The running mechanism of GAs

ture model constraints. For example, according to traditional GAs, a feature set {Win, Dynamic, LRU, API, get, BTree, Unindexed} could be randomly selected from the feature model shown in Figure 1 and be used as an initial population member. However, this random feature set is not a valid product configuration because only one feature from "BTree" and "Unindexed" can be selected. Thus randomly generating initial solutions tends to produce a large set of invalid starting points, which decreases the probability that a valid or optimized solution will be found. In order to develop a good GA for feature selection, developers must devise methods for handling the random generation of a set of valid feature selections. Section 6.3 describes how we address this challenge by applying an algorithm of transforming randomly generated feature sets into valid feature selections.

5.2. Challenge 2: Generating Valid Feature Selections During Feature Selection Evolution

After a GA has generated an initial population, it proceeds to select population members to combine to produce new solutions. A core tenet of a GA is that combining two solutions has the potential to yield another good or better solution. When population members represent feature selections, however, combining two arbitrary solutions without violating feature model constraints is hard.

For example, suppose there are two initial valid solutions for the feature model shown in Figure 1, {DB, OS, BufferMgr, Storage, Win, InMemory, API, Indexing, get, Unindexed} and {DB, OS, BufferMgr, Storage, NutOs, Persistent, API, Indexing, MemAlloc, PageRepl, get, BTree, Dynamic, LRU}. Using the chromosome encoding process described in Section 6.1, we perform a breadth-first traversal (a.k.a., level-order traversal) of the feature model and encode the two initial solutions into two strings, representing their genetic chromosomes: "11101010111001000100000" and "11101101011111100100110". Then we perform a uniform crossover operation (presented in Section 6.6) by a random crossover mask "11001110010011101100 11". Finally we can get a new offspring "11101011011101000101 00" that indicates an invalid feature selection {DB, OS, BufferMgr, Storage, Win, Persistent, API, Indexing, MemAlloc, get, Unindexed, Dynamic}. Section 6.2 describes how we address this challenge by introducing an algorithm, called *fmTransform*, to transform the new generated offspring into a valid feature selection that conforms to the feature model constraints.

5.3. Challenge 3: Generating Feature Selections that Fit Resource Constraints

The SPL feature selection problem becomes more complex when resource constraints, such as CPU $\leqslant 40$, Memory $\leqslant 4096$, and Cost $\leqslant 200$, must be considered. These considerations add further constraints that the population members may violate. Not only may the randomly generated initial population members consume more resources that are available but the evolutionary combination of chromosomes may yield offspring with excess resource consumption.

One common approach used in GAs to handle situations where generated offspring may represent invalid solutions is to use a *repair operator* to fix invalid solutions. A repair operator takes an invalid solution as input and generates a valid solution. The key challenge to applying repair operators to resource consumption violations is that determining how to reach a valid feature selection that satisfies resource constraints from an arbitrary feature selection is an NP-hard problem (White et al., 2009).

Another approach that is commonly employed is to design the objective function so that invalid solutions always score lower than valid solutions. However, designing an objective function to balance resource consumption against desired SPL product capability is hard. Section 6.4 describes how we address this challenge by computing the proportion of the generated value to the consumed resource as the fitness function in GAFES.

6. GAFES: A GA-based AI Approach to Optimized Feature Selection in SPLs

Following the idea of GAs, we designed a GA-based AI approach to automate SPL feature selection optimization with resource constraints, GAFES, which is presented in Algorithm 1. In the following sections, we present a detailed specification of GAFES. The remainder of this section is structured as follows: 1) we define a mathematical representation for the chromosomes (solutions) (Section 6.1); 2) we present an algorithm, called *fmTransform*, that can repair a feature selection that violates the feature model constraints (Section 6.2); 3) we describe how GAFES uses a combination of random feature selection and *fmTransform* to obtain an initial population (Section 6.3); 4) we define the fitness function, present the mechanism of selection and replacement, and give the termination conditions for GAFES (Section 6.4); 5) we define the genetic operations including crossover and mutation (Section 6.6); and 6) we discuss how we determine the initial parameter values for GAFES, including the size of the population, the maximum generation, the crossover probability and the mutation rate.

6.1. Feature Chromosome Encoding

A chromosome in our GAFES represents a feature combination in a feature model. For a feature model with *n* features, each chromosome is composed of *n* genes and each gene represents a feature. Here, a string with *n* binary digits is used to encode a chromosome. A binary digit represents a feature, values 1 and 0 meaning selected and unselected. For example, a chromosome "11101010111001000100000" means that the features {DB, OS, BufferMgr, Storage, Win,

Algorithm 1: GAFES

```
Input: a feature model with $F = \{f_i\}, 1 \leqslant i \leqslant n, C$,
       $R(F)$, $V(F)$, and $Rc$.
Output: $S \subseteq 2^F$.
1  Initialize population $P$;
2  repeat
3      select two parent chromosomes $p1$ and $p2$
       from $P$;
4      offspring = crossover($p1$, $p2$);
5      mutation(offspring);
6      fmTransform(offspring);
7      if $R(offspring) \leqslant Rc$ then
8          replace(P, offspring);
9  until stopping condition ;
10 return $S$ that is the fittest chromosome in $P$;
```

InMemory, API, Indexing, get, Unindexed} are selected, if a breadth-first traversal of the feature model shown in Figure 1 is performed.

6.2. *fmTransform: An Algorithm for Arbitrary Feature Set Transformation*

As is described in Section 5, the key to adapting GAs to the SPL feature selection problem is to develop a mechanism to transform an arbitrary (maybe invalid) feature selection into a valid feature combination. We designed an algorithm, called *fmTransform* to achieve the task. As is shown in Algorithm 2, the input is a feature model with its features F and its constraints C, and an arbitrary feature set S_R. Generally S_R is generated randomly. The output is a valid feature combination S_V transformed from S_R according to C. S_E contains all the features that should not be selected.

Algorithm 2 works as follows. For each feature $f \in S_R$, if $f \notin S_V$ and $f \notin S_E$, then f is included in S_V. Meanwhile, a path passing through the feature f is extended until one end of the path reaches the root feature and the other reaches a terminal feature. All the features in the path are also included in S_V. The above process is repeated until all the features in S_R are traversed. After traversing S_R, we traverse all the features in S_V, find out those features whose children have not yet been included in S_V and put them into S_G. For each feature in S_G, we generate its path from the root feature to a terminal feature and then include all the features through the path in

Step	Input feature in S_R	Initial operation	S_V	S_E
1	Win	Includes('Win')	Win, OS, DB, BufferMgr, Storage, API, Indexing	NutOS
2	Dynamic	Includes('Dynamic')	(+) Dynamic, MemAlloc, Persistent, PageRepl,	(+) Static, InMemory
3	LRU	Includes('LRU')	(+) LRU	(+) LFU
4	API	/	(=)	(=)
5	get	Includes('get')	(+) get	(=)
6	BTree	Includes('BTree')	(+) BTree	(+) Unindexed
7	Unindexed	/	(=)	(=)

(+) means the current result equals to the result of previous step plus the following elements.
(=) means the current result equals to the result of previous step.

Figure 4: Demonstration of executing *fmTransform*

```
Algorithm 2: fmTransform($S_R$)
Input: $F$, $C$, $S_R \subseteq 2^F$.
Output: $S_V \subseteq 2^F$.
1  $S_V \leftarrow \emptyset; S_E \leftarrow \emptyset; S_G \leftarrow \emptyset$;
2  foreach feature $f \in S_R$ do
3      if $f \notin S_V$ and $f \notin S_E$ then IncludeFeature($f$);
4  end
5  foreach feature $f \in S_V$ do
6      if every child feature of $f \notin S_V$ then $S_G \leftarrow f$;
7  end
8  foreach feature $f \in S_G$ do
9      while $f$ is not a terminal feature do
10         $f \leftarrow$ randomly select a child feature of $f$;
11         IncludeFeature($f$);
12     end
13 end
```

S_V, which guarantees the final feature combination complete and valid.

Figure 4 demonstrates a sample process of executing *fmTransform* on the feature model show in Figure 1. The input S_R is {Win, Dynamic, LRU, API, get, BTree, Unindexed}. When a feature in S_R is input, a set of features required by the input feature and the feature model constraints is included in S_V, and another set of features excluded by the input feature and the constraints is included in S_E. The final output S_V is {DB, OS, Win, BufferMgr, Persistent, MemAlloc, Dynamic, PageRepl, LRU, Storage, API, get, Indexing, BTree}, and S_E is {NutOS, Static, InMemory, LFU, Unindexed}.

```
Algorithm 3: IncludeFeature($f$)
1  if $f \notin S_V$ and $f \notin S_E$ then $S_V \leftarrow f$;
2  if $f$ is not the root feature then
3      IncludeFeature(the parent feature of $f$);
4  if $f \in$ an Alternative-group then
5      ExcludeFeature(all $f$'s brother features in the
       same group)
6  if $f$'s children $\in$ an And-group then
7      foreach feature $f' \in$ the group of $f$'s children do
8          if $f'$ is mandatory then IncludeFeature($f'$);
9      end
10 if $f$ excludes $f' \in C$ or $f'$ excludes $f \in C$ then
11     ExcludeFeature($f'$);
12 if $f$ requires $f' \in C$ then IncludeFeature($f'$);
```

```
Algorithm 4: ExcludeFeature($f$)
1  if $f \notin S_V$ and $f \notin S_E$ then $S_E \leftarrow f$;
2  foreach feature $f' \in$ the group of $f$'s children do
   ExcludeFeature($f'$);
3  if $f \in$ an And-group and $f$ is mandatory then
4      ExcludeFeature(the parent feature of $f$);
5  if $f'$ requires $f \in C$ then ExcludeFeature($f'$);
```

Algorithm 3 and Algorithm 4 determine whether a feature should be included ($\in S_V$) or be excluded ($\in S_E$). They works in terms of the following rules. First, if a feature is included and is not the root feature, then its parent should also be included; if a feature is excluded, then its children should also be excluded. Second, for an Alternative-group, if one of its member feature f is included, then all of f's brother features in the same group should be excluded. Third, for an And-group, if its parent feature is included, then all of its mandatory feature should be included; if one of its mandatory feature is excluded, then its parent feature should be excluded. Fourth, for a constraint *A requires B*, if A is included, then B should be included; if B is excluded, then A should be excluded. Fifth, for a constraint *A excludes B*, whatever A or B is included, the other one should be excluded. Sixth, for other features (e.g., the features in an Or-group), if necessary, any feature is selected randomly in order to generate a complete feature combination.

Since a feature model with cross-tree constraints can encode an arbitrary satisfiability problem, it is not always possible to find a valid feature selection. The *fmTransform* algorithm uses a retry counter to limit the time spent attempting to repair a feature selection. In practice, we have found it rare for *fmTransform* to be unable to generate a valid feature selection, but more research is needed to identify feature model architectures for which it does not perform well.

6.3. Initial Population

The initial population represents a set of initial solutions. GAFES' algorithm for generating the initial population is shown in Algorithm 5. A string with n binary digits is randomly generated to represent a chromosome. Each randomly generated chromosome essentially represents a random feature set in a feature model. By applying the algorithm *fmTransform*, these random feature sets are then transformed into a set of valid feature combinations that conform to the feature model constraints. As is described in Section 5.1, all the transformed feature combinations must also satisfy the resource constraints, which means that some initial solutions may be invalid starting points. To prevent the initial population generation step from running indefinitely, GAFES' controls the parameter *retries* to generate an initial population in a limited retry time. Moreover GAFES controls the parameter d to obtain the expected number of selected features in every generated chromosome. Here, the function $random(1)$ generates a random float number within $[0, 1]$.

Algorithm 5: Initial population generation

```
1  $i = 1$;
2  repeat
3      $retries = 0$;
4      while $retries < 2 * ||P||$ do
5          foreach gene $g$ in $chromosome_i$ do
6              if $random(1) < d/n$ then $g = 1$ else
                   $g = 0$;
7          end
8          fmTransform($chromosome_i$);
9          if $R(chromosome_i) \leqslant Rc$ then
10             $i++$; break;
11         else
12             $retries++$; continue;
13         end
14     end
15 until $i > ||P||$ ;
16 sort all chromosomes in $P$ nonincreasingly in
   terms of their fitness;
```

6.4. Feature Selection Fitness Evaluation

In order to evaluate the solutions and select the best one, a fitness metric is needed. For the SPL feature selection optimization with resource constraints, the obvious approach is to allow a developer to supply a domain-specific objective function that describes the value of a solution. The challenge, however, is that directly employing this type of function does not penalize solutions that violate resource constraints. As we described in Section 5.3, this is a significant problem.

To overcome this challenge, GAFES transforms the objective function, supplied by the developer, in order to penalize solutions which overconsume resources. The basic heuristic that GAFES uses to transform the objective function is that the best solutions will produce more value per unit of resource consumption. Thus the fitness of a chromosome Ch is defined as the value of the solution produced by the domain-specific objective function divided by the net aggregate resource consumption. More formally, GAFES' objective function is defined as:

$$Fitness(Ch) = V(Ch)/R(Ch). \tag{4}$$

where:

- **Ch** - is a chromosome in the population, indicating a feature selection in a feature model.
- **V(Ch)** - is the value of the solution *Ch*, determined by the domain-specific objective function provided by the developer. For example, the value produced by a specific feature selection indicates the possible performance, cost, maintainability, stability, etc.
- **R(Ch)** - is the sum of the resources consumed by all the features included in the chromosome *Ch*. The resources consumed by a specific feature indicates the possible CPU, memory, development staff time, etc.

Developers can adapt the fitness function to optimize for different measurement like performance, maintenance, etc. by changing *V(Ch)*. $R(Ch)$ can be seen as a penalty factor of the fitness function, which makes solutions that provide higher value per unit of resource consumption preferred. When facing multiple kinds of resource, $R(Ch)$ can be designed as a multiple-dimensional vector where each dimension indicates one kind of resource. This approach also permits weighting of resources to reflect importance.

6.5. Selection and Replacement of Feature Selections

The chromosome selection process for the next generation adopts a simple random selection mechanism. For each generation, two parent chromosomes are selected randomly in the population. The crossover operation generates a new chromosome (offspring) out of the two parents, and the mutation operation slightly perturbs the offspring.

GAFES adopts the replacement mechanism developed in (Bui & Moon, 1996). If the generated offspring is superior to both parents, it replaces the similar parent; if it is in between the two parents, it replaces the inferior parent; otherwise, the most inferior chromosome in the population is replaced. GAFES stops when the number of generations reaches the predefined maximum generation G.

6.6. Crossover and Mutation

GAFES uses the standard crossover and mutation operations. As is shown in Figure 5, a uniform crossover operation is used to combine bits sampled uniformly from the two parents. In this case, the crossover mask is generated as a random bit string with each bit chosen at random and independent of the others. The point mutation operation produces small random changes to the bit string by choosing a single bit at random, then changing its value. Mutation is performed after crossover.

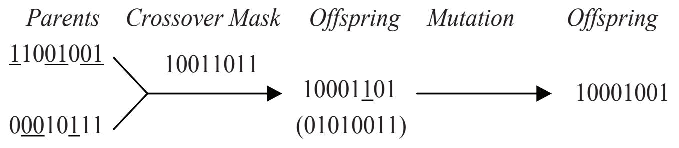

Figure 5: An example of uniform crossover and point mutation

6.7. Parameters

No systematic parameter optimization process has so far been attempted, but we use the following parameters in our experiments: population size $\|P\|$= 30, maximum generation G= 100, crossover probability= 1 (always applied), and mutation rate= 0.1.

6.8. Algorithmic Complexity

We first analyze the algorithmic complexity of *fmTransform*. As is shown in Algorithm 3 and Algorithm 4, both *IncludeFeature(f)* and *ExcludeFeature(f)* cost $O(cm \log n)$, where m is the average number of child features for each feature and c is the maximum number of cross-tree constraints (i.e., all the *requires* and *excludes* relationships) in the feature model. Thus, in Algorithm 2, step 2~4 costs $O(\|S_R\| * (cm \log n))$. Step 5~7 costs $O(\|S_V\| * m)$. Step 8~13 costs $O(\|S_G\| * cm \log^2 n)$. Since $\|S_R\| < n$, $\|S_V\| < n$, and $\|S_G\| < n$, the total time of Algorithm 2, $T(fmTransform)$, is $O(cmn \log n + mn + cmn \log^2 n) = O(cmn \log^2 n)$.

The algorithmic complexity of GAFES shown in Algorithm 1 can be decomposed as follows. The first step requires $O(\|P\| * \log \|P\| + \|P\|^2 * T(fmTransform))$= $O(\|P\|^2 * T(fmTransform))$ time to generate an initial population, as is shown in Algorithm 5. The parents selection operation in step 3 costs $O(1)$, the crossover operation in step 4 $O(n)$, the mutation operation in step 5 $O(1)$, the replace operation in step 8 $O(\|P\|)$. Thus, step 2~9 costs $O(G * (\|P\| + n + T(fmTransform)))$=

$O(G * T(fmTransform))$. Step 10 costs $O(||P||)$. Here, $||P||$=30 and G=100. Thus, the total time of Algorithm 2 is $O(||P||^2 * T(fmTransform) + G * T(fmTransform))$= $O(||P||^2 * T(fmTransform))$= $O(||P||^2 * cmn \log^2 n)$. If there are no cross-tree constraints, the complexity is reduced to $O(||P||^2 * mn \log^2 n)$. Both algorithmic complexities are polynomial.

7. Empirical Results

In this section, we present empirical results from our experiments and evaluate our approach. As an approximation algorithm, our approach cannot guarantee that the generated approximation answer is optimal. For effectiveness evaluation, we introduce an important metric, *optimality* (Approximation Answer/Optimal Answer), to measure how close the algorithm can get to the optimal answer. Another important consideration is the *time consumption* and *scalability* because we must ensure that our approach is efficient even if there are hundreds or thousands of features.

7.1. Experimental Setup

We first adopt Thum's method (Thum et al., 2009) to randomly generate feature models with different characteristics. We also use the test pattern generation technique devised by (Akbar et al., 2001) to generate random optimization problem instances for which we knew the optimal answer. Then we parametrically control the size of feature models for a thorough runtime evaluation.

Independent parameters in our experiments are the number of features in a feature model. The time needed to generate an approximation answer and the optimality are measured as dependent variables. To reduce the fluctuations in the dependent variables caused by random generation, we performed 100 repetitions for each configuration of independent parameters, i.e., we generated 100 random feature models with the same parameters and each performed the same optimization problem. All measurements were performed on the same Windows XP with Inter Pentium 4 CPU 3.0GHz and 2GB RAM.

7.1.1. Feature Model Generation

The algorithm to randomly generate feature models of size n is as follows (Thum et al., 2009): starting with a single *root* node, runs several iterations of the generation process. In each iteration, an existing node without children is randomly selected, and between one and ten children are randomly added. These child nodes are connected to the parent either by And- (50% probability), Or- (25% probability) or Alternative-group (25% probability). Children in an And-group are optional by a 50% probability. This iterative process is continued until the feature model has n features. All features with children are considered non-terminal. Moreover, we also generate cross-tree constraints (*requires* and *excludes*). For every 10 features, one constraint is generated by the following algorithm: two different features are randomly selected, and then are connected randomly by *requires* (50% probability) or *excludes* (50% probability) link. According to Thum's survey (Thum et al., 2009), these parameters are backed up by most of the surveyed feature models and represent a rough average. Thus, these generated feature models basically reflect the characteristics of realistic feature models.

The above generated feature models can be easily translated into propositional formulas (Batory, 2005; Thum et al., 2009). We use the SAT solver *sat4J*[1] to validate these feature models and discard all feature models that do not have a single valid configuration (mostly due to poorly chosen cross-tree constraints). We repeat the entire process until the appropriate number of valid feature models is generated.

We fixed the following parameters: maximum number of children= 10; type of child group= (50%, 25%, 25%); optional child= 50%; number of cross-tree constraints= 0.1*n; variables in cross-tree constrains= 2. Again, Thum's survey (Thum et al., 2009) shows that these parameters reflect real feature models.

7.1.2. Test Pattern Generation

The optimization problem in product derivation is initialized by the following pseudo random numbers:

[1]`http://www.sat4j.org`

resource consumed by each terminal feature $r(tf_i) = random(R_{max})$, resource consumed by each non-terminal feature $r(nf_i) = 0$; value per unit resource $unitV = random(UnitV_{max})$; value for each terminal feature $v(tf_i) = r(tf_i) * unitV + random(V_{max})$, value for each non-terminal feature $v(nf_i) = 0$. Here, R_{max}, $UnitV_{max}$, and V_{max} are the upper bound of resource requirement, unit price of resource, and the extra value of a feature after its consumed resource price. The value of each item is not directly proportional to the resource consumption. The function $random(i)$ returns an integer from 0 to $(i - 1)$, following the uniform distribution. The total resource constraint $Rc = R_{max} * n_{TF} * 0.8$ where n_{TF} is the number of all terminal features in a feature model.

According to (Akbar et al., 2001), if we want to generate a problem for which an optimal answer is known, the following reinitializations have to be done. If S is the set of features selected by an approximation algorithm, then $R(S) = \sum_{f_i \in S} r(f_i)$, i.e, exactly equal to the sum of consumed resources of all selected features, and the value associated with each selected feature is $v(f_i) = r(f_i) * unitV + V_{max}$. Thus $V(S) = \sum_{f_i \in S} v(f_i) = \sum_{f_i \in S}(r(f_i) * unitV + V_{max})$. This reinitializations ensures maximum value per unit resource for the selected feature.

7.2. *Experiment 1: Small Feature Models*

We implemented GAFES using Java. For comparison, we implemented White's method (White et al., 2009), called *Filtered Cartesian Flattening* (FCF), that can transform the SPL feature selection optimization with resource constraints into a Multi-dimensional Multiple-choice Knapsack Problem (MMKP). Then we solved the MMKP problem to generate an optimized feature selection using two kinds of optimization techniques: the Branch and Bound with Linear Programming (BBLP) (Alsuwaiyel, 1999) and the Modified Heuristic (M-HEU) algorithms (Akbar et al., 2001). M-HEU is a heuristic technique. It puts an upper limit on the number of upgrades and downgrades that can be performed (Akbar et al., 2001). BBLP can find an exact solution. But finding exact solutions is NP-hard, which means that it is not feasible to apply BBLP to all practical cases (e.g., $n \geqslant 500$). Therefore, we first performed an experiment on small feature models whose size n varies from 10 to 200.

Hypothesis. Our hypothesis was that GAFES would be faster than FCF+BBLP because GAFES is an approximation algorithm and FCF+BBLP an exact one. GAFES would be also faster than FCF+M-HEU due to general knowledge of traditional GAs. In addition, we believed GAFES could obtain better optimality than the other two approaches.

Experiment 1 Results. Table 2 shows a comparison among FCF+BBLP, FCF+M-HEU, and GAFES. Here, the experiment is performed on each feature model whose size n varies from 10 to 200, as is listed in column 1. 100 feature models for each size are generated randomly with the same parameters, as is described in section 7.1.1, and we only take the mean value of their results. We used the constants $R_{max} = 10$, $UnitV_{max} = 10$, and $V_{max} = 20$ for generation of test cases, as is explained in section 7.1.2. Data is not initialized for a predefined maximum total value for the selected features.

The columns T_{BBLP}, T_{M-HEU}, and T_{GA} show the time requirement for the FCF+BBLP, FCF+M-HEU, and GAFES solutions. The column T_{GAInit} gives the time requirement for the initialization of GAFES. The column T_{GAInit}/T_{GA} presents the proportion of the initialization to the whole GAFES in time consumption. We can see that the initialization time occupies a considerable proportion (about 30-45%) of the whole time.

The columns $(T_{BBLP} - T_{GA})/T_{BBLP}$ and $(T_{M-HEU} - T_{GA})/T_{M-HEU}$ indicate the proportion of the time saved by GAFES to the whole time consumption by FCF+BBLP and FCF+M-HEU respectively. For tiny feature models with 10 features, GAFES consumed more time than FCF+BBLP and FCF+M-HEU. For feature models whose size varies from 50 to 200, GAFES reduced 94-99% time consumption of FCF+BBLP and 92-97% time consumption of FCF+M-HEU.

The column V_{BBLP} gives the average value earned from the FCF+BBLP solutions. The columns V_{M-HEU}/V_{BBLP} and V_{GA}/V_{BBLP} indicate the average standardized value earned in the two heuristics (i.e., FCF+M-HEU and GAFES) with respect to the exact solutions generated by FCF+BBLP. FCF+M-HEU obtains 98-99% optimality while GAFES 87-

Table 2: Experimental results for small feature models

n	T_{BBLP} (ms)	T_{M-HEU} (ms)	T_{GAInit} (ms)	T_{GA} (ms)	$\frac{T_{GAInit}}{T_{GA}}$	$\frac{T_{BBLP}-T_{GA}}{T_{BBLP}}$	$\frac{T_{M-HEU}-T_{GA}}{T_{M-HEU}}$	V_{BBLP}	$\frac{V_{M-HEU}}{V_{BBLP}}$	$\frac{V_{GA}}{V_{BBLP}}$	$\frac{V_{GA}}{V_{M-HEU}}$	σ_{M-HEU}	σ_{GA}
10	0.95	0.79	1.03	3.15	0.3270	-2.3158	-2.9873	348.37	0.986	0.866	0.878	0.0685	0.0699
50	202.99	160.35	3.64	12.27	0.2967	0.9396	0.9235	1362.05	0.989	0.874	0.884	0.0540	0.0402
100	1813.74	1217.44	12.39	31.79	0.3897	0.9825	0.9739	2211.03	0.981	0.903	0.920	0.0356	0.0439
200	8800.76	3070.32	45.77	101.53	0.4508	0.9885	0.9669	3825.20	0.976	0.895	0.917	0.0498	0.0582

90%, which did not beat FCF's optimality as expected. The column V_{GA}/V_{M-HEU} shows that GAFES produces solutions with 88-92% of the optimality of FCF+M-HEU. In addition, σ_{M-HEU} and σ_{GA} are the standard deviation of standardized total value achieved in the 100 feature models, given to indicate stability.

7.3. *Experiment 2: Large Feature Models*

From Table 2, we can see that the time requirements of FCF+BBLP solutions increase dramatically with the size of feature models because of the exponential computation complexity of exact techniques. It is impractical to test the performance of FCF+BBLP solutions for larger feature models. Hence, in this experiment, we only performed FCF+M-HEU and GAFES on large feature models whose size varies from 500 to 10000. Moreover, due to the lack of the solutions of exact techniques (i.e., FCF+BBLP), we adopt the test pattern generation technique described in section 7.1.2 to estimate the optimality achieved by the two heuristics (i.e., FCF+M-HEU and GAFES).

Hypothesis. Based on the results of Experiment 1, our hypothesis for this experiment was that GAFES would be faster than FCF+M-HEU. GAFES would have better scalability than FCF+M-HEU. But, based on the first experiment, we expected GAFES to have slightly lower optimality than FCF+M-HEU.

Experiment 2 Results. Table 3 shows a comparison between FCF+M-HEU and GAFES. Like Experiment 1, 100 feature models for each size are generated randomly with the same parameters, as is described in section 7.1.1, and we only take the mean value of their results. We still used the constants R_{max} = 10, $UnixV_{max}$ = 10, and V_{max} = 20 for generation of test cases, as is explained in section 7.1.2. Data is not initialized for a predefined maximum total value for the selected features.

The columns T_{M-HEU} and T_{GA} show the time requirement for the FCF+M-HEU and GAFES solutions. From the column T_{GAInit}/T_{GA}, we can see that the time requirement for GAFES initialization, T_{GAInit}, accounts for 20-23% of the total time of GAFES. The column $(T_{M-HEU} - T_{GA})/T_{M-HEU}$ shows that GAFES reduces 45-94% time consumption of FCF+M-HEU.

The column $MaxValue$ indicates the average value generated by the reinitialization in the test pattern generation technique. Taking $MaxValue$ as the optimal answer, FCF+M-HEU obtains 89-93% optimality while GAFES about 86%. The column V_{GA}/V_{M-HEU} shows that GAFES produces solutions with 93-97% of the optimality of FCF+M-HEU. From the columns σ_{M-HEU} and σ_{GA}, we can see that all the values, achieved in the 100 random feature models for each size, maintains relatively stable.

7.4. *Discussion of Results & Threats to Validity*

From Table 2 and Table 3, we can conclude that GAFES performs more efficiently than existing exact (i.e., FCF+BBLP) and heuristic (i.e., FCF+M-HEU) approaches. That is, it produced nearly as optimal results (within 86-90%) in 45-99% less time. Only for tiny feature models with 10 features, GAFES consumed more time than FCF+BBLP and FCF+M-HEU, which is caused by the fixed retry time for generating an initial population and the fixed number of maximum generation defined in GAFES. For most of the feature models whose size varies from 20 to 10000, GAFES reduces 45-99% time consumption of FCF+BBLP or FCF+M-HEU. Especially for large feature models (size > 500), GAFES obtains better scalability than FCF+M-HEU. As for optimality, GAFES stays well above 86% optimality, which is with 88-97% optimal of FCF+M-HEU. In addition, the stability of the solution performance is almost same in both the cases generated by the two heuristics.

Table 3: Experimental results for large feature models

n	T_{M-HEU} (ms)	T_{GAInit} (ms)	T_{GA} (ms)	$\frac{T_{GAInit}}{T_{GA}}$	$\frac{T_{M-HEU}-T_{GA}}{T_{M-HEU}}$	$MaxV$	$\frac{V_{M-HEU}}{MaxV}$	$\frac{V_{GA}}{MaxV}$	$\frac{V_{GA}}{V_{M-HEU}}$	σ_{M-HEU}	σ_{GA}
500	6193.43	88.63	384.42	0.2306	0.9379	348.37	0.933	0.863	0.925	0.0530	0.0512
1000	14083.91	354.55	1513.68	0.2342	0.8925	1362.05	0.905	0.862	0.952	0.0699	0.0539
2000	33270.84	1453.25	6211.57	0.2340	0.8133	2211.03	0.897	0.861	0.960	0.0780	0.0484
5000	91429.64	6213.59	31358.35	0.1981	0.6570	3825.20	0.890	0.856	0.962	0.0769	0.0506
10000	194093.20	22486.81	107632.03	0.2089	0.4455	77470.71	0.887	0.857	0.966	0.0715	0.0523

Threats to internal validity are influences that can affect the time requirement of all exact and heuristic solutions that have not been considered. We cannot guarantee that computation time depends on certain shapes of a feature model or certain resource constraints. However, to avoid effects of certain feature models, all of input feature models are generated automatically by simulating known feature models and each measurement is repeated 100 times with freshly generated feature models.

Threats to external validity are conditions that limit our ability to generalize the results of our experiments to industrial practice. We generated feature models with the described algorithm and parameters, and confirmed that they align well with those known feature models acquired from existing publications in SPL community. We cannot guarantee that our generated test cases are typical in practice. However, to simulate practical situation as far as possible, we randomly generate the consumed resource and the provided value for each terminal features. Moreover, test data is not initialized for a predefined maximum total value for the selected features. The total resource constraint is generated dynamically for each random feature model.

8. Related Work

This section compares our work on GAFES to related work on AI approaches and other approaches for feature selection optimization, including exact and heuristic techniques and visualization techniques.

8.1. Exact Techniques for SPL Feature Selection

Many exact techniques in the AI community have been applied and extended to solve the SPL feature selection optimization. Benavides et al. (Benavides et al., 2005) considered resource constraints in product derivation process and applied Constraint Satisfaction Problems (CSPs) to model and solve feature selection problems automatically. Their technique works well for small-scale problems where an approximation technique is not needed. For large-scale problems, however, their technique is too computationally demanding. In contrast, GAFES works well on large-scale problems.

Mannion (Mannion, 2002), Batory (Batory, 2005), and Czarnecki et al. (Czarnecki & Wasowski, 2007) applied propositional logic to automated feature selection. However, these techniques were not designed to handle integer resource constraints and thus cannot handle the SPL feature selection optimization with resource constraints. Moreover, these techniques depend on SAT or BDD solvers that use exponential algorithms. GAFES is a polynomial-time algorithm that can handle integer resource constraints and thus can solve the SPL feature selection optimization with resource constraints even on large-scale problems.

Zhang et al. (Zhang et al., 2003) poposed a technique for reasoning about SPL variant quality attributes using Bayesian Belief Networks. Jarzabek et al. (Jarzabek et al., 2006) developed another technique in this area based on soft goals. However, these techniques is designed for selecting features in situations where it is difficult to predict the impact of a feature selection. But in GAFES, the exact impact of each feature selection on the resource consumption of the variant is known. The two techniques are complementary to each other. If the exact impact of feature selections on variant quality is not known, the techniques of Zhang or Jarzabek can be used. If the impact of feature selection is known, GAFES is appropriate.

8.2. Heuristics for SPL Feature Selection

Although exact algorithms to solve NP-hard problems have exponential complexity, each NP-hard problem typically has a number of approximation algorithms that can be used to solve it with acceptable optimality. Search-based software engineering (Harman, 2007) advocates the application of optimization techniques from the operations research and heuristic (or metaheuristic) computation research communities to software engineering. It has already had several successful application domains in software engineering (Harman, 2007), such as optimizing the search for requirements to form the next release (Bagnall et al., 2001) and optimizing test data selection and prioritization (Walcott et al., 2006). However, to the best of our knowledge, there is only one work from White et al. (White et al., 2009) providing an approximation algorithm for the SPL feature selection optimization with resource constraints.

White et al. (White et al., 2009) provided a polynomial time approximation algorithm for selecting a highly optimal set of features that adheres to a set of resource constraints. They proposed the FCF technique that transforms the optimized feature selection problem into the MMKP problem, and then use the M-HEU technique (Akbar et al., 2001) to solve the MMKP problem. However, their approach still requires significant computing time for large-scale problems. Moreover their approach involves two approximation processes, one is in the FCF stage to transform all And- and Or-groups into Alternative-groups, another is in the M-HEU stage to generate an approximation solution. In addition, their approach demands a higher resource tightness, which is also a common problem for the heuristic techniques that solve the MMKP problem (Akbar et al., 2001; White et al., 2009).

GAFES also provides a heuristic solution. Compared with White's method (FCF+M-HEU), it has only one approximation process, i.e, the genetic process of generating an approximation solution. Moreover GAFES avoids the limitation of resource tightness because it transforms the SPL feature selection optimization into a genetic problem not an MMKP problem. Experiments show that GAFES can reduce 45-97% time consumption of FCF+M-HEU.

8.3. Feature Model Visualization Techniques

Some researchers developed special visualization techniques to assist developers in decision making during the product derivation process. Sellier and Mannion (Sellier & Mannion, 2007) proposed a visualization metamodel for representing inter-dependencies between SPLs and described a tool to realize these visualizations. Botterweck et al. (Botterweck et al., 2007) presented a metamodel that described staged feature configuration and introduced a tool that illustrated the advantages of interactive visualization in managing feature configuration. However, these approaches focus more on functional requirements of a product and their dependencies and less on non-functional requirements or resource constraints (Siegmund et al., 2008). Moreover, industrial-sized feature models with hundreds or thousands of features make a fully manual feature selection process usually impossible.

9. Concluding Remarks & Lessons Learned

To quickly derive an optimal product configuration, developers need algorithmic techniques to automatically generate a valid feature selection that optimize desired product properties. However, finding a feature selection with optimal product capability subject to required constraints is an NP-hard problem. Although there are numerous heuristic techniques for many NP-hard problems, they cannot directly support the SPL feature selection optimization with resource constraints because they are not designed to handle various structural and semantic constraints defined in the feature model. This lack of heuristics limits the scale of feature model on which developers can realistically optimize and evaluate a large number of possible product configurations.

This paper presents a GA-based feature selection approach, GAFES, for automated product derivation in SPLs. The key to adapting GAs to the SPL feature selection optimization successfully is that we design an algorithm that can transform an arbitrary feature set into a valid feature combination conforming to the feature model constraints. Experiments show that GAFES can achieve an average of 86-90% optimality even for large feature models. Moreover GAFES can derive a feature selection 45-99%

faster than existing heuristic and exact feature selection techniques.

From our research on GAFES, we learned the following important lessons:

1. For large-scale feature models, exact techniques suffer from exponential algorithmic complexity and typically require days, months, or more, to solve the SPL feature selection optimization with resource constraints. In general, heuristic techniques have a polynomial algorithmic complexity. GAFES is heuristic and achieves 45-97% faster than existing heuristics (e.g., FCF+M-HEU) for the SPL feature selection problem.
2. GAFES produces solutions with 86-97% optimal of prior feature selection techniques. However, prior feature selection techniques, such as FCF+BBLP and FCF+M-HEU, obtain better optimality at the cost of longer solving time, which is caused by traversing larger solution space. In addition, the optimality gained by GAFES maintains relatively stable for different size of feature models.
3. GAFES is most applicable to domains, such as distributed real-time and embedded systems where the precise impact of feature selections on the performance and resource consumption of the product is known.
4. A key component of GAFES is the algorithm *fmTransform* that can, in most cases, transform an arbitrary feature set into a valid feature combination in a feature model. Since many stochastic heuristics, such as GAs and Ant Colony Optimization techniques, work on a range of randomly generated feature sets and finally generate an arbitrary feature selection, the algorithm *fmTransform* can be seen as a trigger for applying these stochastic heuristics to the SPL feature selection problem.

In future work, we plan to adapt more heuristics, such as ant colony optimization (Al-Ani, 2005) and particle swarm optimization (Lin et al., 2008), to the SPL feature selection optimization with resource constraints.

Acknowledgments

Funding was provided by the National Natural Science Foundation of China (NSFC No. 60773088), the National High-tech R&D Program of China (863 Program No. 2009AA04Z106), and the Key Program of Basic Research of Shanghai Municipal S&T Commission (No. 08JC1411700).

References

Akbar, M. M., Manning, E. G., Shoja, G. C., Khan, S., 2001. Heuristic Solutions for the Multiple-Choice Multi-dimension Knapsack Problem. In: Proceedings of ICCS'01, San Francisco, CA, USA, pp. 659-668.

Al-Ani, A., 2005. An Ant Colony Optimization Based Approach for Feature Selection. In: Proceedings of AIML'05, Cairo, Egypt.

Alsuwaiyel, M. H., 1999. Algorithms: design techniques and analysis. World Scientific.

Bagnall, A. J., Rayward-Smith, V. J., Whittley, I., 2001. The next release problem. Information & Software Technology 43 (14), 883-890.

Batory, D., 2005. Feature Models, Grammars, and Propositional Formulas. In: Proceedings of SPLC'05, Rennes, France, pp. 7-20.

Batory, D., Benavides, D., Ruiz-Cortes, A., 2006. Automated analysis of feature models: challenges ahead. Communications of the ACM 49 (12), 45-47.

Benavides, D., Martin-Arroyo, P. T., Cortes, A. R., 2005. Automated Reasoning on Feature Models. In: Proceedings of CAiSE'05, Porto, Portugal, pp. 491-503.

Benavides, D., Ruiz-Cortes, A., Batory, D., Heymans, P., 2008. First International Workshop on Analysis of Software Product Lines (ASPL'08). In: Proceedings of SPLC'08, Limerick, Ireland, pp. 385.

Boeing Company, 2002. Bold Stroke Avionics Software Family. URL: <http://splc.net/fame/boeing.html>.

Botterweck, G., Nestor, D., Preubner, A., Cawley, C., Thiel, S., 2007. Towards Supporting Feature Configuration by Interactive Visualisation. In: Proceedings of 1st International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2007), Tokyo, Japan, pp. 125-131.

Bui, T. N., Moon, B. R., 1996. Genetic Algorithm and Graph Partitioning. IEEE Transactions on Computers 45 (7), 841-855.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA, USA.

Czarnecki, K., Wasowski, A., 2007. Feature Diagrams and Logics: There and Back Again. In: Proceedings of SPLC'07, Kyoto, Japan, pp. 23-34.

Deelstra, S., Sinnema, M., Bosch, J., 2004. Experiences in Software Product Families: Problems and Issues During Product Derivation. In: Proceedings of SPLC'04, Boston, MA, USA, pp. 165-182.

Deelstra, S., Sinnema, M., Bosch, J., 2005. Product derivation in software product families: a case study. Journal of Systems and Software 74 (2), 173-194.

Dudley, G., Joshi, N., Ogle, D. M., Subramanian, B., Topo, B. B., 2004. Autonomic Self-Healing Systems in a Cross-Product IT Environment. In: Proceedings of ICAC'04, New York, NY, USA, pp. 312-313.

FAME-DBMS, Project Prototypes. URL: <`http://fame-dbms.org/index.shtml`>.

Guo, J., Wang, Y., 2010. Towards Consistent Evolution of Feature Models. In: Proceedings of SPLC'10, Jeju Island, South Korea, pp. 451-455.

Harman, M., 2007. The Current State and Future of Search Based Software Engineering. In: Proceedings of Workshop on the Future of Software Engineering (ISCE'07), Minneapolis, MN, USA, pp. 342-357.

Jarzabek, S., Yang, B., Yoeun, S., 2006. Addressing quality attributes in domain analysis for product lines. Software, IEE Proceedings 153, 61-73.

De Jong, K. A., Spears, W. M., 1989. Using Genetic Algorithms to Solve NP-Complete Problems. In: Proceedings of ICGA'89, George Mason University, Fairfax, Virginia, USA, pp. 124-132.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., Peterson, A. S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, CMU.

Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., 1998. FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5 (1), 143-168.

Kang, K. C., Lee, J., Donohoe, P., 2002. Feature-oriented product line engineering. IEEE Software 19 (4), 58-65.

Lin, S.-W., Ying, K.-C., Chen, S.-C., Lee, Z.-J., 2008. Particle swarm optimization for parameter determination and feature selection of support vector machines. Expert Systems with Applications 35 (4), 1817-1824.

Loesch, F., Ploedereder, E., 2007. Optimization of Variability in Software Product Lines. In: Proceedings of SPLC'07, Kyoto, Japan, pp. 151-162.

Mannion, M., 2002. Using First-Order Logic for Product Line Model Validation. In: Proceedings of SPLC'02, San Diego, CA, USA, pp. 176-187.

Mendonca, M., Branco, M., Cowan, D. D., 2009. S.P.L.O.T.: software product lines online tools. In: Proceedings of OOPSLA, Orlando, Florida, USA, pp. 761-762.

Mitchell, M., 1996. An introduction to genetic algorithms. MIT Press, Cambridge, MA.

Muhlenbein, H., 1989. Parallel Genetic Algorithms Population Genetics and Combinatorial Optimization. In: Proceedings of ICGA'89, George Mason University, Fairfax, Virginia, USA, pp. 416-421.

Oh, I.-S., Lee, J.-S., Moon, B. R., 2004. Hybrid Genetic Algorithms for Feature Selection. IEEE Transactions on Pattern Analysis and Machine Intelligence 26 (11), 1424-1437.

Pohl, K., Bockle, G., van der Linden, F., 2005. Software Product Line Engineering: Foundations, Principles, and Techniques. Springer-Verlag, Berlin Heidelberg.

Rosenmuller, M., Siegmund, N., Schirmeier, H., Sincero, J., Apel, S., Leich, T., Spinczyk, O., Saake, G., 2008. FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In: Proceedings of EDBT Workshop on Software Engineering for Tailor-made Data Management, Nantes, France, pp. 1-6.

Sellier, D., Mannion, M., 2007. Visualizing Product Line Requirement Selection Decision. In: Proceedings of 2nd International Workshop on Requirements Engineering Visualization (REV'07), New Delhi, India.

Siegmund, N., Rosenmuller, M., Kuhlemann, M., Kastner, C., Saake, G., 2008. Measuring Non-Functional Properties in Software Product Line for Product Derivation. In: Proceedings of APSEC'08, Beijing, China.

Steger, M., Tischer, C., Boss, B., Muller, A., Pertler, O., Stolz, W., Ferber, S., 2004. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In: Proceedings of SPLC'04, Boston, MA, USA, pp. 34-50.

Thum, T., Batory, D. S., Kastner, C., 2009. Reasoning about edits to feature models. In: Proceedings of ICSE'09, Vancouver, Canada, pp. 254-264.

Walcott, K. R., Soffa, M. L., Kapfhammer, G. M., Roos, R. S., 2006. Time aware test suite prioritization. In: Proceedings of ISSTA'06, Portland, Maine, USA, pp. 1-12.

White, J., Dougherty, B., Schmidt, D. C., 2009. Selecting highly optimal architectural feature sets with Filtered Cartesian Flattening. Journal of Systems and Software 82 (8), 1268-1284.

Yang, J. H., Honavar, V., 1998. Feature Subset Selection Using a Genetic Algorithm. IEEE Intelligent Systems 13 (2), 44-49.

Zhang, H., Jarzabek, S., Yang, B., 2003. Quality Prediction and Assessment for Product Lines. In: Proceedings of CAiSE'03, Klagenfurt, Austria, pp. 681-695.