Search-based Diverse Sampling from Real-world Software Product Lines

Yi Xiang
xiangyi@scut.edu.cn
South China University of Technology
Guangzhou, China

Han Huang*
hhan@scut.edu.cn
South China University of Technology
Guangzhou, China

Yuren Zhou
School of Computer Science and Engineering, Sun Yat-Sen University
Guangzhou, China

Sizhe Li
South China University of Technology
Guangzhou, China

Chuan Luo
Microsoft Research
Beijing, China

Qingwei Lin
Microsoft Research
Beijing, China

Miqing Li
University of Birmingham
Birmingham, UK

Xiaowei Yang*
xwyang@scut.edu.cn
South China University of Technology
Guangzhou, China

ABSTRACT

Real-world software product lines (SPLs) often encompass enormous valid configurations that are impossible to enumerate. To understand properties of the space formed by all valid configurations, a feasible way is to select a small and valid sample set. Even though a number of sampling strategies have been proposed, they either fail to produce diverse samples with respect to the number of selected features (an important property to characterize behaviors of configurations), or achieve diverse sampling but with limited scalability (the handleable configuration space size is limited to 10^{13}). To resolve this dilemma, we propose a scalable diverse sampling strategy, which uses a distance metric in combination with the novelty search algorithm to produce diverse samples in an incremental way. The distance metric is carefully designed to measure similarities between configurations, and further diversity of a sample set. The novelty search incrementally improves diversity of samples through the search for novel configurations. We evaluate our sampling algorithm on 39 real-world SPLs. It is able to generate the required number of samples for all the SPLs, including those which cannot be counted by sharpSAT, a state-of-the-art model counting solver. Moreover, it performs better than or at least competitively to state-of-the-art samplers regarding diversity of the sample set. Experimental results suggest that only the proposed sampler (among all the tested ones) achieves scalable diverse sampling.

CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**; • **Mathematics of computing** → **Optimization with randomized search heuristics**.

KEYWORDS

Software product lines, diverse sampling, novelty search, distance metric

ACM Reference Format:
Yi Xiang, Han Huang, Yuren Zhou, Sizhe Li, Chuan Luo, Qingwei Lin, Miqing Li, and Xiaowei Yang. 2022. Search-based Diverse Sampling from Real-world Software Product Lines. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510053

1 INTRODUCTION

Software product lines (SPLs) [11], being highly configurable, allow users to derive products by selecting and deselecting *features*, which are increments of product functionality. That is, a set of features defines a unique product (or *configuration*) of an SPL. Clearly, as the number of features increases, the number of all possible configurations grows exponentially [41]. A common tool for representing all valid configurations is a tree-like structure, called a *feature model* (FM) [32], in which features and constraints among them are explicitly specified. The space formed by all valid configurations is called a *configuration space*, denoted as Ψ henceforth. In practice, because large real-world SPLs often encompass (hundreds of) thousands of features, the size of Ψ, i.e., $|\Psi|$,

*Corresponding authors.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510053

could be astronomically large ($\gg 10^{10}$) [43]. Therefore, it is rarely possible to enumerate every valid configuration. To achieve a certain goal, e.g., learning a performance prediction model [31], testing SPLs [45], we need to sample from the configuration space a small set of valid configurations, called a *sample set*, where each valid configuration is known as a *sample*.

Quite often, sample sets must be well-chosen based on domain knowledge. For instance, a sample set should cover all t-wise feature combinations in the context of *t-wise* sampling [2, 19, 30]. If no domain knowledge is available, however, sample sets are expected to cover the configuration space as widely and uniformly as possible [31, 43]. There have been several sampling strategies in the literature, e.g., *random sampling* [20, 22, 37–39], *solver-based sampling* [10, 18, 24, 26], *coverage-oriented sampling* [2, 19, 30] and *uniform sampling* [1, 9, 40, 42, 43, 49, 51]. These sampling strategies focus on different aspects of sampling from SPLs, and come with different strengths and weaknesses (detailed discussions are available in Section 6).

In this paper, we focus on another kind of sampling, known as *diverse sampling*, which seems to be largely ignored. Diverse sampling brings lots of benefits. For example, it could reduce the risk of missing important configurations with distinct performance behavior when deriving a performance prediction model (see [31]), and it forms a scalable and flexible alternative to t-wise sampling (see [25, 54]). Recently, Kaltenecker et al. [31] proposed a diverse sampling strategy, called *diversified distance-based sampling* (DDbS), which pursues to derive diverse samples regarding *the number of selected features*. In fact, the number of selected features for a configuration $c \in \Psi$, denoted as $\mathcal{T}(c)$, is important to characterize the behavior of this configuration. Pursuing diversity in terms of this number can directly improve the accuracy of performance prediction, which has been shown in [31]. Moreover, the number of selected features is directly treated as an optimization goal in multi-objective SPL configuration [24, 28, 50, 57]. In such scenarios, a common and important issue is how to improve the samples' diversity regarding this number.

Though important, achieving diverse sampling regarding the number of selected features poses great challenges to state-of-the-art samplers. Figs. 1 (a) and (b) show the distribution of the number of selected features for samples generated by two recent uniform samplers (i.e., Smarch [43] and Unigen3 [51]) on the HiPAcc feature model [31]. As seen, compared with DDbS [31] [see Fig. 1 (c)], both of them are unable to produce diverse samples concerning the number of selected features. Notice that even though DDbS could sample more diverse configurations on this model, it faces the scalability issue. In fact, DDbS failed to handle configuration space larger than 10^{13} [43]. This is also confirmed by our experiments performed in Section 5.3, and explained later in Section 5.5. In summary, these samplers either fail to produce diverse configurations, or achieve diverse sampling but with limited scalability.

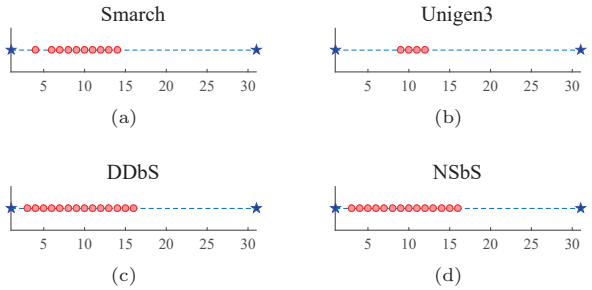

Figure 1: Distribution of the number of selected features for samples generated by four samplers on HiPAcc [31]. In this figure, the x-axis is the number of selected features in a configuration; $\star$ denotes the estimated boundaries of the above number (see Section 2.2). Note that these boundaries may not be reachable. We pursue to cover the range between the two boundaries as diversely as possible.

To enable scalable diverse sampling, this paper provides an alternative perspective, i.e., search-based sampling. The key idea is to generate initial samples using efficient off-the-shelf SAT solvers, and then incrementally improve the diversity of the sample set using a specific search algorithm. This sampling strategy relies on a special distance metric and a search technique, called novelty search (NS) [34, 35]. We name this sampling algorithm *NS-based sampling* (NSbS for short). To demonstrate merits of NSbS, we compare it with several state-of-the-art sampling algorithms using 39 real-world SPLs adopted by Oh et al. [43]. Experimental results reveal that NSbS indeed enables a scalable diverse sampling from SPLs. In particular, it successfully generates the requested number of configurations (i.e., 100 configurations in our setting) for all 39 SPLs, including the largest ones on which most state-of-the-art samplers fail to generate even one configuration within an hour.

Main contributions of the paper are summarized as follows.

- A tailored distance metric. By using the number of selected features, the configuration space Ψ is mapped to a small behavior space $\mathcal{B} = \{\mathcal{T}(c)|c \in \Psi\}$[1]. A distance metric is designed to measure similarities between configurations in both $\mathcal{B}$ and Ψ. We show, both theoretically and experimentally, that using this distance metric not only improves the coverage in the behavior space, but also promotes diversity in the original configuration space. Considering diversity in both spaces could improve the representativeness of the sample sets.
- A befitting search technique. We choose NS as the search engine because of its good theoretical properties [16, 17] that well fit the goal of diverse sampling.

[1]The behavior space is a concept introduced in NS algorithms [34, 35]

Precisely, NS has been shown to tend towards a diverse/uniform sampling of the behavior space [16, 17]. The above property could help to improve diversity of the sample set in the behavior space. As shown in Fig. 1 (d), samples generated by NSbS are as diverse as those of DDbS, which, as mentioned early, is a tailored diverse sampler.

- Flexibility in the sampling process. Since diversity is improved in an incremental way, it is easy for users to achieve a desired trade-off between diversity and efficiency. If a higher-quality sample set is required, then more execution time can be specified. The flexibility in the sampling process is one of the main advantages of NSbS over other state-of-the-art samplers, most of which are not controllable regarding the execution time.

2 PRELIMINARIES

In this section, we provide necessary preliminaries on sampling from configuration spaces, and the space mapping strategy. Moreover, a brief introduction to NS is also presented.

2.1 Sampling from configuration spaces

Formally, an FM can be seen as a tuple $\langle \mathcal{F}, \mathcal{C} \rangle$, where $\mathcal{F} = \{f_1, \ldots, f_n\}$ is the set of n features, and $\mathcal{C}$ is the set of all constraints among features. A *configuration* c, represented by $\{\pm f_1, \ldots, \pm f_n\}$, is defined as a set of selected or deselected features. Precisely, $+f_i$ and $-f_i$ indicate that the feature f_i is selected and deselected, respectively. In programming, c can be represented by a binary string, with 1 indicating a selected feature, and 0 a deselected one. Due to constraints in $\mathcal{C}$, not all configurations are valid. The configuration that satisfies all the constraints is called a valid configuration, and all valid configurations form the configuration space Ψ.

Many software engineering tasks require to derive a small sample set from Ψ. In this context, a *sample set*, $\boldsymbol{S} = \{s_1, \ldots, s_N\}$ (where N denotes the sample size), is a subset of Ψ, i.e., $\boldsymbol{S} \subseteq \Psi$. Manually deriving samples is error-prone and time-consuming even for tiny FMs. Therefore, automated solvers, like SAT solvers, have been widely adopted to generate samples from Ψ [25, 36, 57]. It is well-known that an FM can be easily converted into a propositional formula ϕ [6]. The derived ϕ is then used as the input of automated solvers, which are internally run to find solutions to ϕ.

2.2 Mapping to behavior spaces

As mentioned previously, by characterizing the behavior of a configuration using the number of selected features, $c \in \Psi$ is mapped to an integer. Accordingly, the configuration space Ψ is mapped into the behavior space $\mathcal{B}$. Let Φ_b be the space formed by all configurations with exactly $b \in \mathcal{B}$ selected features, then $\Psi = \cup_{b \in \mathcal{B}} \Phi_b$. That is to say, the whole configuration space Ψ is decomposed into $|\mathcal{B}|$ subspaces. We can then sample configurations that are diversely distributed among these subspaces.

Understanding $\mathcal{B}$ is much easier than Ψ due to that $|\mathcal{B}|$ is significantly smaller than $|\Psi|$. In fact, the lower and upper bounds for $\mathcal{B}$ can be approximated by using the number of core features (denoted by $|core|$) and the number of dead features (denoted by $|dead|$), respectively. Notice that core features must be selected in every valid configuration, while dead features must not be selected. To be more specific, $\min(\mathcal{B}) \geq |core|$, and $\max(\mathcal{B}) \leq n - |dead|$, where n is the total number of features. Therefore, the size of $\mathcal{B}$ is at most $n - (|core| + |dead|) + 1$. In contrast, $|\Psi|$ grows exponentially with respect to n. Hence, Ψ can be astronomically large, especially for large real-world SPLs. For example, $|\Psi|$ is as large as 7.78×10^{417} for the uClinux-config model [43]. We must mention that, in theory, knowing exactly $\mathcal{B}$ is as hard as knowing Ψ because every configuration should be investigated in the worst case. However, $\mathcal{B}$ can be well approximated by the following set, $\mathcal{B}' = \{|core|, |core| + 1, \ldots, n - |dead|\}$, which contains all possible integers from $|core|$ to $n - |dead|$. It is possible that there exist some integers to which no configurations are mapped. Therefore, $\mathcal{B}$ is a subset of $\mathcal{B}'$. In Section 4.3, $\mathcal{B}'$ will be used to calculate performance indicators.

2.3 Novelty search

As mentioned in Section 1, NS [34, 35] is adopted in our search-based diverse sampling. Therefore, it is necessary to give a brief introduction to this search technique. NS is one of the main divergent search algorithms [34], and its prominent feature is to abandon objectives [34]: it replaces the conventional goal-oriented objective by a criterion measuring *novelty* of individuals. This criterion is referred to as *novelty score*, defined as the average distance of an individual to its k nearest neighbors, where k is a constant. Formally, $\rho(x)$, the novelty score of x, is given as follows [16, 34, 35].

$$\rho(x) = \frac{1}{k} \sum_{j=1}^{k} d(x, x_j) \tag{1}$$

where x_j is the j-th nearest neighbor of x among an archive of previously explored individuals and the current population in the behavior space; $d(x, x_j)$ is any distance metric. $\rho(x)$ estimates the sparseness of x in the behavior space. If this score is large, then x is in a sparse area; in contrast, it is in a dense area in case that the novelty score is small. In general, individuals in sparse regions are preferred to those in dense regions as the exploitation around sparse regions is helpful to perform a diverse exploration of the behavior space [16].

Following the practice in [54], the calculation of the novelty score can be extended from a single configuration to a sample set $\boldsymbol{S} = \{s_1, \ldots, s_N\}$. Specifically,

$$\rho(\boldsymbol{S}) = \sum_{i=1}^{N} \rho(s_i) \tag{2}$$

where $\rho(s_i)$ is the novelty score of a single configuration, as given by Eq. (1). Clearly, the higher the novelty score, the more diverse the sample set.

Finally, it is worth mentioning that there is an interesting search behavior of NS. That is, *the sampling produced by NS covers the whole reachable behavior space* [16]. This suggests that NS explores the behavior space diversely. It is a good property, which well matches the goal of diverse sampling from SPLs. Therefore, we choose NS as the search engine.

3 NS-BASED DIVERSE SAMPLING

The NSbS procedure is outlined in Algorithm 1. The key idea is to continuously improve diversity of the initial sample set through the search for *novel* individuals (i.e., configurations). The algorithm takes the propositional formula ϕ (derived from a given FM) and the sample size N as input, and outputs a set of samples stored in an archive $\mathcal{A}$.

Algorithm 1: NSbS algorithm

```
Input: φ (propositional formula), N (sample size)
Output: A (archived samples)
1  Initialize the archive A by generating N solutions to φ
   using the randomized SAT4J solver [25];
2  Initialize the distance matrix D = (d_ij)_{(N+1)×(N+1)},
   where d_ij (i, j = 1, ..., N), as given in Eq. (3), is the
   distance between x_i ∈ A and x_j ∈ A;
3  For each x ∈ A, calculate its novelty score ρ(x) based
   on Eq. (1);
4  while the termination condition is not met do
5      {p1, p2} ← matingSelection(A);
6      {c1, c2} ← crossover(p1, p2);
7      for i ∈ {1, 2} do
8          c_i ← mutation(c_i);
9          if c_i is invalid then
10             Repair c_i using the probSAT solver [5];
11         end
12         A ← updateAchive(A, c_i);
13     end
14 end
15 return A
```

3.1 Initialization

As shown in Line 1 of Algorithm 1, $\mathcal{A}$ is initialized with N configurations generated by the randomized SAT4J solver [7], in which the order how the logical clauses and the literals are parsed is randomized [24]. According to the implementation in [24, 26], there exist three parsing strategies, i.e.,*NegativeLiteralSelectionStrategy*, *PositiveLiteralSelectionStrategy* and *RandomLiteralSelectionStrategy*. Each strategy has an equal chance of being chosen when generating initial configurations. In particular, the first strategy prefers negative assignments to literals, and thus emphasizes configurations with less selected features. Therefore, the lower bound of $\mathcal{B}$ can be approximated by using this strategy. Similarly, the second strategy helps to approximate the upper bound of $\mathcal{B}$. The third strategy randomly assigning *true* or *false* to literals is able to improve randomness of the generated configurations. Using simultaneously three strategies aims at improving diversity of the initial sample set. In particular, bounds of $\mathcal{B}$ could be well approximated. We should mention that this SAT-based seeding, instead of random seeding, is used here because the former always generates valid configurations while the latter is highly likely to generate unwanted invalid ones due to the constraints.

3.2 Distance metric

To measure similarities between two configurations, we define the following distance[2]:

$$d_{ij} = d(x_i, x_j) = \frac{1}{2} \cdot \left(\frac{abs\left(\mathcal{T}(x_i) - \mathcal{T}(x_j)\right)}{n} \right) + \frac{1}{2} \cdot \left(1 - \frac{|x_i \cap x_j|}{n} \right) \cdot \delta \tag{3}$$

where $x_i, x_j \in \mathcal{A}$ ($x_i \neq x_j$) are two different configurations; $\mathcal{T}(x_i)$ denotes the number of selected features in x_i; $abs(\cdot)$ returns the absolute value of a number; and $|\cdot|$ returns the cardinality of a set. δ is a constant, as given below.

$$\delta = \begin{cases} \frac{1}{\max\{\mathcal{T}(x_i), \mathcal{T}(x_j)\}} & \mathcal{T}(x_i) + \mathcal{T}(x_j) \leq n, \\ \frac{1}{n - \min\{\mathcal{T}(x_i), \mathcal{T}(x_j)\}} & \text{otherwise.} \end{cases} \tag{4}$$

As seen, this distance metric consists of two weighted parts. The first part measures the similarity between configurations in the behavior space, while the second part in the original configuration space. In fact, $1 - \frac{|x_i \cap x_j|}{n}$ is the Hamming distance [3] between x_i and x_j. Note that using the above two parts is intended to sample configurations covering diversely in the behavior space, and also keeping as dissimilar as possible in the configuration space. In Section 5.1, we will experimentally verify this distance metric.

It is also worth noting that δ is set based on our theoretical analysis, which shows that δ is needed to mitigate biases towards sampling specific configurations. In other words, Eq.(3) without using δ can introduce biases in the behavior space, and thus can hamper diversity of the sample set. Detailed analysis can be found in Section S-1 of the supplement[3]. In Section 5.2, we will experimentally investigate δ's effects. Therein, one will find that using δ indeed improves diversity of the sample set in the behavior space.

According to Line 2 in Algorithm 1, the distance matrix $\mathbf{D}_{(N+1)\times(N+1)}$ is initialized by working out the distance between each pair of configurations in $\mathcal{A}$. Note that, since $\mathbf{D}$ is symmetric, we only need to calculate distances for half of these pairs. We would like to mention that the size of $\mathbf{D}$ is $(N+1) \times (N+1)$, rather than $N \times N$, because we reserve spaces for storing distances when evaluating a new configuration (see Algorithm 2). After obtaining $\mathbf{D}$, as shown in Line 3 of Algorithm 1, the novelty score for each $x \in \mathcal{A}$ is calculated based on Eq. (1).

[2] If $x_i = x_j$, d_{ij} is forcibly set to 0.

[3] The online supplement is available at https://doi.org/10.5281/zenodo.5828178

3.3 Genetic operations

Like in genetic algorithms, we perform in order the mating selection, crossover and mutation to generate new individuals. As shown in Line 5 of Algorithm 1, the *matingSelection* procedure chooses from $\mathcal{A}$ two parents p_1 and p_2 each time. The basic idea of choosing a parent is to select the one with larger novelty score from two different random members in $\mathcal{A}$. In case of a tie, a random selection is performed between the two members. Clearly, the above mating selection emphasizes individuals located in sparse regions. Exploration around sparse regions could potentially improve diversity of the samples.

Algorithm 2: $\mathcal{A} \leftarrow updateAchive(\mathcal{A}, c)$

Input: $\mathcal{A}, c$
Output: $\mathcal{A}$
1 **if** $\mathcal{A}$ contains c **then**
2 **return** $\mathcal{A}$;
3 **end**
4 **for** $i = 1, \ldots, N$ **do**
5 $d_{i,(N+1)} \leftarrow d(x_i, c)$;
6 $d_{(N+1),i} \leftarrow d_{i,(N+1)}$;
7 **end**
8 $d_{(N+1),(N+1)} \leftarrow 0$;
9 For each $x \in \mathcal{A} \cup c$, calculate its novelty score $\rho(x)$ based on Eq. (1);
10 $x_{worst} \leftarrow \arg\min_{x \in \mathcal{A}} \rho(x)$ // Find the worst member in $\mathcal{A}$;
11 **if** $\rho(c) > \rho(x_{worst})$ **then**
12 $x_{worst} \leftarrow c$;
 `// Update D`
13 **for** $j = 1, \ldots, N$ **do**
14 $d_{j,worst} \leftarrow d_{j,(N+1)}$;
15 $d_{worst,j} \leftarrow d_{(N+1),j}$;
16 **end**
17 $d_{worst,worst} \leftarrow 0$;
18 **end**
19 **return** $\mathcal{A}$

Once two parents p_1 and p_2 have been selected, the uniform crossover is applied to generate two children, c_1 and c_2 (Line 6 in Algorithm 1). To be specific, for each index $j \in \{1, \ldots, n\}$, we generate a random number *rand*. If $rand < 0.5$, then $c_1(j)$ and $c_2(j)$ are set to $p_1(j)$ and $p_2(j)$, respectively. Otherwise, they are set to $p_2(j)$ and $p_1(j)$, respectively. Notice that $c_1(j)$ denotes the value taken in the j-th position of c_1. The newly generated individuals are then subjected to bit-wise mutation (Line 8 in Algorithm 1). Specifically, for each bit, the value is changed from 1 (true) to 0 (false), or vice versa. Often, the ratio of bits to be changed is controlled by a parameter P_μ, called mutation probability. In this work, we set P_μ to 0.1, following the common practice in [55].

It is not uncommon that the resulting configurations (after crossover and mutation) are invalid. In this case, as shown in Line 10 of Algorithm 1, the probSAT solver [5], one of the high-performing stochastic local search (SLS) SAT solvers, is adopted to repair invalid configurations. The variables to be flipped by the solver are chosen based on probabilities such that more promising variables are given more chances to be selected. In fact, probSAT [5] has been adopted to repair infeasible configurations in prior work [56, 57] in the context of optimal products selection from SPLs. In particular, the empirical study in [56] suggested that probSAT is more effective than WalkSAT [8], another popular SLS solver, in improving diversity of a configuration set. For more details on probSAT, we direct readers to the original study [5]. Notice that internal parameters of this solver are set following the practice in [5] and [56]. Therefore, a tuning phase is not required in this work.

Configurations operated by probSAT could still be invalid (even though they are valid most of the time)[4], in particular for large-scale FMs. In case of invalidity, we simply request to the randomized SAT solver, as described in Section 3.1, to return a valid configuration.

3.4 Updating archive

The archive $\mathcal{A}$ stores novel configurations discovered during the search process. Its update procedure is presented in Algorithm 2. To improve diversity, as shown in Lines 1-3, the producer rejects the entrance of any configuration that is identical to already archived ones. When a totally different configuration c is available, we need to fill the distance matrix $\mathbf{D}$ by working out distances between c and each $x_i \in \mathcal{A}$. These distances are stored in the last row and the last column. In what follows, as indicated in Line 9 of Algorithm 2, the novelty score for each member $x \in \mathcal{A} \cup c$ is calculated based on Eq. (1). We should note here that the novelty scores are computed taking into account not only members in $\mathcal{A}$ but also the new configuration c. This enables an evaluation of the novelty with respect to both previously explored individuals and the current one that represents the most recently visited point [34].

In Line 10 of Algorithm 2, we find the worst member from $\mathcal{A}$, and this member is denoted by x_{worst}, where the index *worst* is its position in the archive. In case that the novelty score of c is higher than that of x_{worst}, we will replace x_{worst} by c. Subsequently, the distance matrix should be updated. This is achieved by simply copying the last row (column) to the *worst*-th row (column) (see Lines 13-16). At last, $d_{worst,worst}$ should be set to 0.

According to the above update procedure, the algorithm consistently looks for novel individuals, pushing individuals to constantly move in the behavior space: *in new and unexplored areas first, but also then in already explored areas as their density of individuals is never exactly homogeneous* [17]. This way, diversity of the samples can be persistently improved.

[4]Different from conflict-driven clause learning solvers [14], such as SAT4J, SLS-type SAT solvers offer no guarantees on finding valid assignments.

3.5 Termination conditions

Termination of NSbS can be flexibly specified by users. We offer the following two termination strategies.

Strategy 1: Termination controlled by the maximum running time (max_t). This is a common way of stopping a search algorithm, and the setting of max_t depends largely on the demands of users.

Strategy 2: Automatic termination when the algorithm gets relatively steady. This is achieved by adding the following piece of codes after Line 13 in Algorithm 1.

If $\frac{|\rho(\widetilde{\mathcal{A}})-\rho(\mathcal{A})|}{\rho(\widetilde{\mathcal{A}})} < 0.1\%$
  $counter++$
Else
  $counter \leftarrow 0$
End

where $\widetilde{\mathcal{A}}$ is the old archive, while $\mathcal{A}$ is the newly updated one. If the change ratio of novelty scores for the two archives is below 0.1%, then *counter* is increased by one; otherwise, it is reset to 0. The algorithm will terminate once *counter* exceeds R, a threshold specified by users. In our experiments, we set R to 10. With this setting, it is found that NSbS automatically terminates on almost all of the tested FMs.

We would like to mention that we give users freedom to specify the termination of the sampling process. Most of the state-of-the-art samplers are not controllable with respect to the execution time. Indeed, it may take excessively long before a set of samples is returned [43, 47]. Instead, the proposed NSbS allows users to make a desired trade-off between quality (primarily diversity) and efficiency. If users want a higher-quality sample set, he/she can set R or max_t to a relatively larger value. The flexibility regarding terminations is one of the main advantages of NSbS over other state-of-the-art samplers.

4 EXPERIMENT SETUP

In this section, we start by introducing our research questions (RQs). Then, we give information about FMs used in our empirical study. Subsequently, we describe how the performance of different samplers can be measured using specialized indicators. Finally, detailed implementations are given.

4.1 Research Questions

The distance metric is expected to play an important role in sampling products that are diverse not only in the behavior space, but also in the original configuration space. It is necessary to investigate the effect of components in Eq. (3). With this regard, we aim at answering the following two RQs.

RQ1: *What are the benefits brought by using the two weighted parts in the distance metric?*

RQ2: *Does the factor δ matter in the distance metric of NSbS?*

To address RQ1, we perform an ablation study where the distance metric defined in Eq. (3) is compared against a modified one in which the second part is removed. Our primary goal is to sample diverse configurations in the behavior space $\mathcal{B}$. On top of this, we also expect that they are as diverse as possible in the configuration space Ψ. The first part in the distance metric is designed for the primary goal, and eliminating it will lead to failure of this goal (because the second part only measures similarity in Ψ). Hence, we only remove the second part in the ablation study. According to our theoretical analysis, the goal of δ in Eq. (3) is to alleviate biases towards specific configurations in the behavior space. The second research question amounts to experimentally verifying this.

Moreover, we intend to answer two more research questions regarding the effectiveness of NSbS in comparison with several state-of-the-art samplers, and impacts of the parameter k.

- **RQ3**: *How effective is NSbS concerning both scalability and diversity in comparisons with state-of-the-art samplers?*
- **RQ4**: *How is the performance of NSbS affected by its key parameter k?*

To address RQ3, we compare NSbS with SAT-based sampling [24], DDbS [31], UniGen3 [51] and Smarch [43]. We expect that NSbS performs better than or at least competitively to them with respect to both scalability and diversity. Finally, the fourth research question seeks to provide useful guidelines for tuning NS in the context of diverse sampling from SPLs.

4.2 Subject Feature Models

In our experiments, we consider 39 FMs that have been carefully selected by Oh et al. [43] in their evaluation of Smarch. Table 1 gives an overview of the subject FMs, including the number of features ($|\mathcal{F}|$), the number of CNF constraints ($|\mathcal{C}|$), the size of the configuration space ($|\Psi|$), the number of core, dead and unconstrained[5] features (i.e., $|core|$, $|dead|$ and $|uc|$). Note that $|\Psi|$ is counted by sharpSAT [53], which fails on the last five largest FMs. All FMs are publicly available in DIMACS format[6], the standard format for SAT solvers.

4.3 Performance indicators

Performance indicators are required to evaluate the quality of a sample set $\boldsymbol{S} = \{s_1, \ldots, s_N\}$. To measure whether $\boldsymbol{S}$ widely covers the behavior space $\mathcal{B} = \{b_1, \ldots, b_{|\mathcal{B}|}\}$, motivated by the definition of inverted generational distance [12], the following indicator, which we call *Spread*, is defined.

$$Spread(\boldsymbol{S}, \mathcal{B}) = \frac{1}{|\mathcal{B}|} \sum_{i=1}^{|\mathcal{B}|} d_{min}(b_i, \boldsymbol{S}), \tag{5}$$

where $d_{min}(b_i, \boldsymbol{S})$ denotes the minimum distance from b_i to $\boldsymbol{S}$. Mathematically, $d_{min}(b_i, \boldsymbol{S})$ is in the following form

$$d_{min}(b_i, \boldsymbol{S}) = \min_{j=1}^{N} abs(b_i - \mathcal{T}(s_j)). \tag{6}$$

[5]Unconstrained features here are those that are not involved in the CNF constraints.

[6]All FMs are downloaded from https://github.com/jeho-oh/Smarch

Table 1: Overview of the subject feature models

FM	$\|\mathcal{F}\|$	$\|\mathcal{C}\|$	$\|\Psi\|$	$\|core\|$	$\|dead\|$	$\|uc\|$
lrzip	20	63	1.44E+ 02	2	0	0
LLVM	11	1	1.02E+ 03	1	0	10
X264	16	11	1.15E+ 03	3	0	7
Dune	17	16	2.34E+ 03	2	0	0
BerkeleyDBC	18	19	2.56E+ 03	2	0	7
HiPAcc	31	104	1.35E+ 04	1	0	0
JHipster	45	104	2.63E+ 04	7	0	0
Polly	40	100	4.00E+ 04	9	0	0
7z	44	210	6.86E+ 04	5	0	0
JavaGC	39	105	1.93E+ 05	7	0	0
VP9	42	104	2.16E+ 05	9	0	0
fiasco_17_10	234	1178	1.00E+ 10	7	6	7
axTLS_2_1_4	94	190	2.00E+ 12	4	0	2
fiasco	1638	5228	3.58E+ 14	49	964	6
toybox	544	1020	1.45E+ 17	4	365	0
axtls	684	2155	4.29E+ 20	3	381	0
uClibc-ng_1_0_29	269	1403	8.00E+ 26	14	0	37
toybox_0_7_5	316	106	1.40E+ 81	8	15	203
uClinux	1850	2468	1.63E+ 91	7	1237	0
ref4955	1218	3099	8.20E+ 123	0	50	0
adderII	1276	3206	3.57E+ 125	0	37	0
ecos-icse11	1244	3146	4.97E+ 125	0	35	0
m5272c3	1323	3297	2.37E+ 125	0	43	0
pati	1248	3266	7.90E+ 126	0	47	0
olpce2294	1274	3881	8.19E+ 126	0	42	0
integrator_arm9	1267	50606	4.06E+ 129	0	44	0
at91sam7sek	1319	3963	9.45E+ 131	0	74	0
se77x9	1319	49937	1.20E+ 135	0	58	0
phycore229x	1360	4026	1.77E+ 136	0	89	0
busybox-1.18.0	6796	17836	8.50E+ 216	12	3939	0
busybox_1_28_0	998	962	1.30E+ 248	12	0	177
embtoolkit	23516	180511	2.10E+ 252	839	6561	1
freebsd-icse11	1396	62163	8.39E+ 313	3	38	50
uClinux-config	11254	31637	7.78E+ 417	15	6012	0
buildroot	14910	45603	/	100	6895	12
freetz	31012	102705	/	117	14445	0
2.6.28.6-icse11	6888	343944	/	58	102	41
2.6.32-2var	60072	268223	/	1100	31960	12
2.6.33.3-2var	62482	273799	/	1200	33233	12

Regarding $\mathcal{B}$, as mentioned in Section 2.1, it is not exactly known, but can be easily approximated by $\mathcal{B}'$. In practice, we therefore calculate $Spread(\boldsymbol{S}, \mathcal{B}')$, instead of $Spread(\boldsymbol{S}, \mathcal{B})$. It is clear that a smaller value of *Spread* indicates a more diverse distribution in the behavior space. In addition to *Spread*, the novelty score of a sample set, $\rho(\boldsymbol{S})$, as given in Eq. (2), also serves as a performance indicator. It measures the diversity of $\boldsymbol{S}$ in the original configuration space.

4.4 Detailed implementations

For each FM, we sample 100 configurations and compute the average sampling time per configuration (measured in milliseconds) to compare efficiency. All samplers except NSbS terminate once 100 configurations are sampled, or the sampling time takes more than 3600,000 milliseconds (i.e., one hour). Since NSbS is able to quickly sample 100 configurations, it terminates either automatically based on **Strategy 2** as described in Section 3.5, or forcibly when the sampling time reaches a timeout of one hour. Note that, to mitigate random bias, all samplers are independently run 30 times, and we present and analyze experimental results regarding mean values of the performance indicators.

All experiments are performed on a Quad Core@2.20 GHz with 8 GB of RAM running Ubuntu 20.04.2. Source codes of SAT-based sampling [24], DDbS [31], UniGen3 [51] and Smarch [43] are downloaded from their authors' repositories, and they are all executed on a single thread (i.e., without parallelization), following the practice in [27]. The codes of NSbS can be found in our repository[7].

5 RESULTS

In this section, we provide a series of experimental results regarding the research questions. Due to limited space, raw results are given in Tables S-1 to S-3 in the online supplement[8]. To determine whether the difference between different algorithms (over all the 30 runs) is significant or not, following guidelines suggested by Arcuri and Briand [4], the Mann-Whitney U test with a 0.05 significance level is performed for each FM. In these tables, test results are represented by three symbols: •, ‡ and ◦, indicating that the algorithm in the first column performs better than, equivalently to and worse than the algorithm in other columns, respectively. In the following subsections, the performance comparisons are based on these results.

5.1 RQ1: Benefits brought by using the two weighted parts in the distance metric

To investigate benefits brought by using two weighted parts in Eq.(3), we consider two different distance metrics. The first one, as given in Eq.(3), uses two weighted parts. Hereafter, we call it *weighted* distance. The second one retains only the first part, and therefore measures only the similarity in the behavior space. This metric is called *unweighted* distance. Both distance metrics are tested within the same NS framework in which $k = 15$. This setting of k has been widely employed in NS-related literature [21]. Our tuning experiments presented in Section 5.4 suggest that $k = 15$ is also a good setting in our context. When the *weighted* distance is used, the sampling algorithm automatically terminates according to **Strategy 2**. In the case of *unweighted* distance, the termination is controlled by **Strategy 1**, in which max_t is set to the running time consumed by the corresponding algorithm using the *weighted* distance. This setting allows us to investigate the benefits while eliminating potential impacts brought by using different running time.

Regarding *Spread*, the *weighted* distance performs significantly better than its counterpart on 3 out of all the 39 FMs, but worse on only one FM, i.e., the simplest LLVM. For all the remaining 35 FMs, the two distance metrics have similar performance. The above results suggest that using alone the first part of Eq. (3) (which measures similarities in the behavior space) is enough to obtain good spread in the behavior space in the majority of the cases.

[7] https://github.com/YiXiangScut/NSbS
[8] https://doi.org/10.5281/zenodo.5828178

Regarding the novelty score, the *weighted* distance shows significant improvements over the *unweighted* one on 26 out of 39 FMs (67%), and degenerations on only lrzip and 2.6.33.3-2var. Clearly, the second part of Eq. (3) is necessary in promoting diversity in the original configuration space.

Therefore, the answer to RQ1 is clear. *Using the two weighted parts in Eq. (3) indeed brings benefits: it improves coverage in the behavior space; at the same time, it also promotes diversity in the original configuration space. Boosting diversity in both spaces is beneficial for enhancing representativeness of the sample set.*

5.2 RQ2: δ matters in the distance metric

According to Section 3.1, δ in Eq. (3) is used to mitigate potential biases towards specific configurations. In this section, we are going to experimentally examine the effects of this factor. To this end, we compare NSbS (described in Algorithm 1) against its variant, i.e., NSbS-δ in which δ is omitted in the distance metric. The only difference between the two algorithms is the presence or absence of δ.

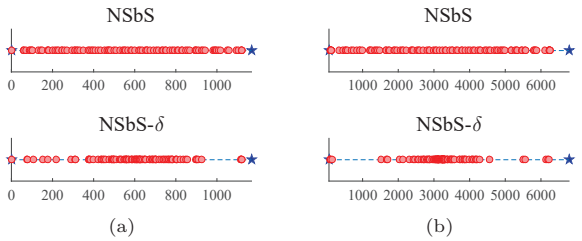

Figure 2: Configurations sampled by NSbS widely cover the behavior space, while those sampled by NSbS-δ fail. (a) ref4955, (b) 2.6.28.6-icse11

As shown by the *Spread* results, NSbS performs better than or at least comparably to NSbS-δ on all the 39 FMs. In particular, NSbS significantly outperforms its counterpart on $25/39 = 64\%$ of the FMs. Moreover, the improvements are mostly observed on large FMs. Taking ref4955 and 2.6.28.6-icse11 as example, Fig. 2 graphically shows the distribution of the sampled configurations in the behavior space. As seen, configurations sampled by NSbS are distributed more widely than those sampled by NSbS-δ on the two FMs. More specifically, configurations of NSbS-δ cover intensively in the middle part, but sparsely at the boundaries.

The above experimental results bring out the following. *The δ indeed matters: when δ is omitted, diversity of the sampled configurations is significantly affected in the behavior space. In particular, boundaries of the behavior space tend not to be sufficiently covered. The above experimental results are in line with our theoretical findings, stating that the adoption of δ is able to mitigate bias towards sampling specific configurations.*

Table 2: Time taken to sample a configuration (in milliseconds). *timeout* = one hour

FM	NSbS	SAT-based	DDbS	Unigen3	Smarch
lrzip	2	1	18	<1	185
LLVM	2	<1	5	<1	110
X264	87	<1	10	<1	151
Dune	2	<1	14	<1	163
BerkeleyDBC	148	<1	14	<1	165
HiPAcc	2	<1	99	5	329
JHipster	10	<1	161	1	388
Polly	92	<1	400	1	373
7z	120	<1	1299	3	467
JavaGC	9	<1	780	1	381
VP9	6	<1	2412	2	396
fiasco_17_10	12	<1	timeout	15	3260
axTLS_2_1_4	6	<1	timeout	12	895
fiasco	93	1	timeout	20	58003
toybox	13	<1	timeout	7	7123
axtls	14	1	timeout	34	13515
uClibc-ng_1_0_29	7	<1	timeout	4681	4239
toybox_0_7_5	7	<1	timeout	231	3492
uClinux	61	2	timeout	258	39713
ref4955	46	1	timeout	timeout	37299
adderII	49	1	timeout	timeout	48508
ecos-icse11	40	1	timeout	timeout	timeout
m5272c3	45	1	timeout	timeout	43946
pati	46	1	timeout	timeout	38404
olpce2294	51	1	timeout	timeout	53946
integrator_arm9	65	2	timeout	timeout	377666
at91sam7sek	49	1	timeout	timeout	45776
se77x9	70	2	timeout	timeout	timeout
phycore229x	49	1	timeout	timeout	56323
busybox-1.18.0	202	3	timeout	timeout	timeout
busybox_1_28_0	37	<1	timeout	timeout	18087
embtoolkit	2306	35	timeout	timeout	timeout
freebsd-icse11	201	7	timeout	timeout	timeout
uClinux-config	296	7	timeout	timeout	timeout
buildroot	6916	21	timeout	timeout	timeout
freetz	16540	35	timeout	timeout	timeout
2.6.28.6-icse11	558	29	timeout	timeout	timeout
2.6.32-2var	36000	256	timeout	timeout	timeout
2.6.33.3-2var	36000	289	timeout	timeout	timeout

5.3 RQ3: Effectiveness of NSbS in comparison with state-of-the-art samplers

Table 2 gives the average time (measured in milliseconds) to sample a single configuration for all FMs. If the sampling can not finish within one hour, then we declare a *timeout*. According to Table 2, SAT-based simpler scales very well, being able to sample one configuration within 300 milliseconds even for the largest FM, i.e., 2.6.33.3-2var. Quite often, the sampling takes no more than 1 millisecond. For DDbS, it can only handle 11 small FMs with $|\Psi| \leq 2.16\times10^5$. For Unigen3, it succeeds in dealing with 19 FMs with $|\Psi| \leq 1.63 \times 10^{91}$. Regarding Smarch, it scales better than DDbS and Unigen3, but still fails on 11 FMs. For our NSbS, it dose not encounter a *timeout* for all FMs. We would like to mention that even though NSbS runs out of one hour on 2.6.32-2var and 2.6.33.3-2var, it successfully samples 100 configurations as requested. Therefore, we do not declare a *timeout*. In fact, this is totally different from the timeout of other samplers,

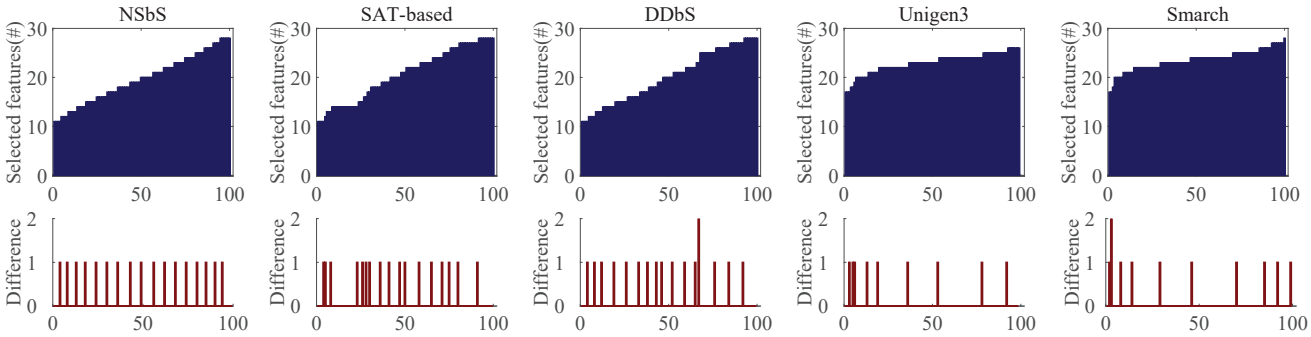

Figure 3: For configurations generated by each sampler on JHipster, the number of selected features and the difference of this number between two successive configurations are shown in histograms.

which are unable to sample 100 configurations within one hour. Regarding the sampling speed, as shown in Table 2, NSbS is slower than the SAT-based sampler, but much faster than Smarch.

Table 3 summarizes Wilcoxon's test results for each pairwise comparison between NSbS and each sampler regarding *Spread*. In this table, available cases refer to those without a *timeout* for both samplers. As can be found, NS performs better than or at least competitively to other samplers in almost all the available cases. The only exception is observed on the pairwise comparison between NSbS and DDbS on the simplest LLVM. In this case, NSbS is significantly worse than DDbS. This exceptional case accounts for 9% of all the available 11 cases for the pair NSbS v.s. DDbS. In summary, NSbS is more effective than SAT-based sampler, Unigen3 and Smarch in generating diverse configurations; NSbS and DDbS are able to sample configurations covering similarly in the behavior space. However, as discussed previously, DDbS suffers from the scalability issue, being only able to handle very small FMs.

Table 3: Summary of Wilcoxon's test results (regarding *Spread*) for pairwise comparisons between NSbS and each sampler

NSbS v.s.	SAT-based	DDbS	Unigen3	Smarch
Available cases (#)	39	11	19	28
•	77%	0%	68%	82%
‡	23%	91%	32%	18%
∘	0%	9%	0%	0%

For configurations generated by the five samplers on JHipster (chosen as an example), we plot in Fig. 3 the number of selected features, and the difference of this number between two successive configurations. Notice that configurations in these sample sets are sorted in increasing order based on the number of selected features. It can be found in Fig. 3 that the number of selected features for NSbS increases more regularly than that for other samplers. To be more specific, the increment of this number for NSbS is steady being always one, while it is either one or two for DDbS and Smarch. In addition, configurations sampled by NSbS can be partitioned into nearly equal-sized subsets based on the number of selected features. For other samplers, however, this partition is less balanced. This can be observed from histograms for the 'difference', indicating that some groups have more configurations than others. The above graphical results suggest that NSbS is capable of sampling configurations that are widely and nearly-uniformly distributed in the behavior space.

Experiments performed in this section emphasize the following. *First, NSbS and SAT-based sampling are the two best samplers regarding scalability, and both of them can handle all FMs under study. Second, NSbS and DDbS perform best concerning diversity of the samples in the behavior space. Therefore, only NSbS (among all samplers tested in this section) achieves scalable diverse sampling.*

5.4 RQ4: Parameter study on k

In NSbS, k is an important parameter which determines how many configurations in the archive are used to calculate the novelty score. To investigate the impact of this parameter, we consider six values for k, i.e., 2, 15, 25, 50, 75 and 100. Notice that 2 is the minimum possible value for k. When $k = 2$, the novelty score of a configuration is evaluated based on its two closest neighbors (including itself). The value 15 has been widely employed in NS-related literature [21], while values 25, 50, 75 and 100 are 1/4, 1/2, 3/4 and 4/4 of the sample size (i.e., 100), respectively. Testing multiple values of k allows us to observe the trend of the performance as k increases. Note that, to eliminate the impacts of initial population, the same set of configurations is initialized for all values of k in each of the 30 independent run.

Fig. 4 presents, in the form of boxplots, *Spread* values over all runs on four representative FMs. It can be found that $k \in \{50, 75, 100\}$ performs significantly worse than $k \in \{2, 15, 25\}$ on all FMs. According to these results, k should be set to relatively small values, e.g., $k < 50$. Furthermore, it can be found that $k = 15$ yields the best performance on most of the feature models. Hence, $k = 15$ is advisable.

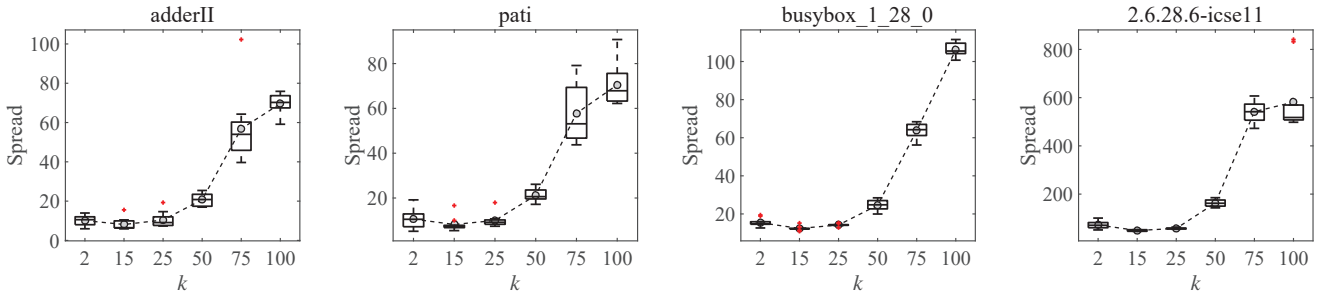

Figure 4: Parameter study on k in NSbS. Based on these results, $k = 15$ is recommended.

Finally, the following conclusions could be drawn based on the above parameter study. *First, the performance of NSbS is indeed affected by its parameter k. In general, small values for k are preferred to large ones. Second, the value 15 is recommended for this parameter. That is to say, in the context of search-based diverse sampling, we can directly set k to the value that has been widely used in NS-related studies [21].*

5.5 Discussions

It is not surprising that NSbS performs better than SAT-based sampler regarding diversity. In fact, initial population in NSbS is the outcome of the SAT-based sampler. This initial population is sequently improved by NS in an incremental way. As shown in Fig. 5, novelty scores of the sample sets are persistently improved during the sampling process of NSbS, naturally leading to more and more diverse samples.

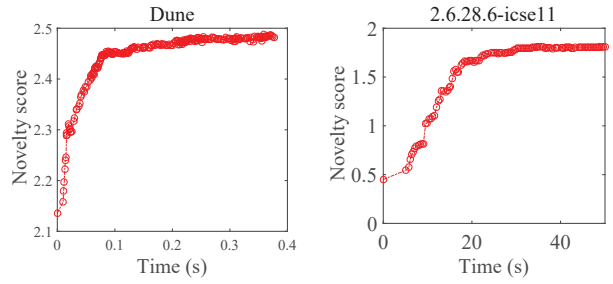

Figure 5: Novelty scores of the sample sets are persistently improved during the sampling process of NSbS

It is also easy to explain why DDbS is computationally much more expensive than NSbS. In fact, NSbS uses genetic operations to generate temporary configurations, and then adopts probSAT [5] to repair them if necessary. This is an efficient way of creating new configurations. Instead, each time DDbS requests to the Z3 constraint solver [15] to find a configuration with exactly d selected features, where d is uniformly drawn from the set of all possible distances. However, it is not always easy to find such configurations because they may not exist. Sometimes the constraint solver takes long to find a feasible configuration, or fails to return any one even after a long time of running. Hence, DDbS suffers from low efficiency.

The following is the reason why the two uniform samplers (Unigen3 and Smarch) can not generate diverse samples in the behavior space. In fact, Unigen3 and Smarch aim at deriving uniform samples in the configuration space. The uniformity in this space cannot guarantee diversity in the behavior space. Recall in Section 2.2 that $\Psi = \cup_{b \in \mathcal{B}} \Phi_b$. If a subspace Φ_b is larger, then more configurations will be sampled from this subspace. All these configurations collapse to a single point in the behavior space. Clearly, this mechanism could hamper diversity in the behavior space.

5.6 Threats to Validity

In this section, we briefly discuss threats to internal validity and external validity, as well as how they could be mitigated.

Internal validity. This type of threats can be caused by potential errors in our implementation of NSbS and the samplers used for comparisons. To rule out errors in the implementation, we have thoroughly tested our codes by analyzing the outcomes step by step on small FMs. For samplers used in performance comparisons, they were implemented by codes provided by their authors.

Due to stochastic nature of the samplers under study, outcomes of different runs could be different. To diminish random biases, we independently run the samplers 30 times, and compare them based on mean values of performance indicators. In addition, statistical tests are utilized to make reliable comparisons.

External validity. This threat is related to the degree to which we can generalize from the experiments. To increase external validity, we select 39 real-world FMs from different domains, and most of them have been widely used by others to evaluate their sampling algorithms [31, 43, 47]. These FMs are representative with respect to the configuration size, which ranges from 10^2 to more than 10^{417}. Therefore, we are confident that our results could generalize to many more FMs.

6 RELATED WORK

There are different strategies for sampling configurations from SPLs: random sampling, solver-based sampling, coverage-oriented sampling and uniform sampling.

Random sampling: The simplest way to create a sample set is to randomly assign *true* or *false* to each feature for each configuration [22, 37–39]. Due to constraints among features, however, this method is very likely to generate invalid configurations. Instead of randomly selecting features, there exist sampling approaches randomly selecting configurations either from all the enumerated configuration space [49], or by using the Monte-Carlo method without exhausted enumeration [20]. Nevertheless, these approaches also select invalid configurations, or suffer from low efficiency because of the time-consuming or even impractical enumeration.

Solver-based sampling: Off-the-shelf SAT or *satisfiability modulo theories* constraint solvers have been widely used to derive samples. These solvers include SAT4J [24, 25, 36], PicoSAT [10, 48], Z3 solver [18, 23, 31, 44]) and stochastic local search SAT solvers [54, 56, 57]. This kind of sampling generally scales well to large real-world SPLs, but offers no guarantees about randomness or coverage [31, 47]. In particular, to improve the diversity of configurations, Henard et al. [24] randomized the order how the logical clauses and the literals are parsed. The resulting randomized SAT4J solver, which has been extensively adopted in different contexts [25, 26, 54], is selected as a baseline in this paper. According to our results, this solver cannot give any guarantees about coverage in the behavior space, though.

Coverage-oriented sampling: It creates a sample set according to a specific coverage criterion. One of the prominent example is *t-wise* sampling in which all possible t feature combinations must be covered [13]. Nowadays, various t-wise sampling approaches are available, e.g., Chvatal [29], ICPL [30], IncLing [2], YASA [33] and CASA [19]. Based on the evaluations in [38], however, most of the t-wise sampling techniques can only deal with small FMs considering often $t = 1$ or $t = 2$. For large real-world SPLs and/or high t interaction strengths, they often run out of memory, do not terminate, or take too much running time [46].

Uniform sampling: Achieving uniform sampling is important to understand properties of the whole configuration space [43]. Recently, uniform sampling has caught increasing attention from both SAT and SPL communities [1, 27, 47]. UniGen2 [9] partitions the configuration space as evenly as possible using hashing functions. Subsequently, sampling is done by choosing a partition at random, and then generating a valid configuration in that partition using an SAT solver. Unigen2 also supports parallelism on sampling, and its improved version, i.e., UniGen3 [51], is now available. Several strategies perform counting-based uniform sampling. Typically, they subsequently partition the configuration space on variable assignments, and then count the number of configurations of the resulting parts. In [40], the number of valid configurations can be easily counted since an FM is encoded as a binary decision diagram. Both *Spur* [1] and *Smarch* [43] rely on sharpSAT [53] to count the number of valid configurations. The above samplers guarantee uniform sampling, but may encounter a bottleneck in some cases. According to Sundermann et al. [52], none of their evaluated model counting solvers, including sharpSAT, can count the number of valid configurations for some large industrial SPLs. Alternatively, QuickSampler [18] performs an efficient sampling of configurations using only a small number of MAX-SAT solver calls. This sampler, however, offers no guarantees on uniformity or even validity of the samples [27, 47].

7 CONCLUSIONS

This paper focuses on diverse sampling from SPLs. In practice, the number of selected features for a configuration is important to characterize its behaviors. By using this number, the configuration space is mapped to a small behavior space. Deriving a small set of valid configurations that has a good coverage in the behavior space is required in many software engineering tasks. However, most existing sampling strategies fail to achieve this goal. In this paper, we propose a search-based sampling strategy which adopts an efficient off-the-shelf SAT solver to generate an initial sample set, and then improves its diversity in an incremental way. This is achieved by using a special distance metric in combination with the novelty search algorithm. Experimental results on 39 real-world SPLs demonstrate that our sampling algorithm can not only improve coverage in the behavior space, but also promote diversity in the original configuration space. Moreover, we show, both theoretically and experimentally, that the designed distance metric is able to mitigate bias towards covering specific parts in the behavior space. Finally, our results show that only the proposed sampling algorithm achieves scalable diverse sampling among all the five evaluated samplers.

Focusing on sampling diverse configurations from behavior spaces, this paper provides a search-based sampler, which is a general tool. Other than the number of selected features, other metrics can also be applicable. In addition, it will be very useful in the future to design dedicated genetic operators in the search algorithm. Currently, we focus on the sampling mechanism itself. As one of the subsequent studies, we will apply the tool to some real-world problems, e.g., t-wise testing [54] and performance prediction [31].

ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation of China (61906069, 61876207), Science and Technology Program of Guangzhou (202002030355, 201802010007), Guangdong Basic and Applied Basic Research Foundation (2019A1515011411, 2019A1515011700), Guangdong Province Key Area R&D Program (2018B010109003), Science and Technology Innovation 2030–"New Generation Artificial Intelligence" Major Project (2020AAA0108404), Fundamental Research Funds for the Central Universities (2020ZYGXZR014), and Microsoft Research Asia.

REFERENCES

[1] Dimitris Achlioptas, Zayd S. Hammoudeh, and Panos Theodoropoulos. 2018. Fast Sampling of Perfectly Uniform Satisfying Assignments. In *Theory and Applications of Satisfiability Testing – SAT 2018*, Olaf Beyersdorff and Christoph M. Wintersteiger (Eds.). Springer International Publishing, Cham, 135–147.

[2] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient Product-Line Testing Using Incremental Pairwise Sampling. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Amsterdam, Netherlands) *(GPCE 2016)*. Association for Computing Machinery, New York, NY, USA, 144–155.

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

[4] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE'11)*. Association for Computing Machinery, New York, NY, USA, 1–10.

[5] Adrian Balint and Uwe Schöning. 2012. *Choosing Probability Distributions for Stochastic Local Search and the Role of Make versus Break*. International Conference on Theory and Applications of Satisfiability Testing, Berlin, Heidelberg, 16–29.

[6] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Conference Software Product Lines, SPLC 2005*, Henk Obbink and Klaus Pohl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 7–20.

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

[8] Shaowei Cai, Chuan Luo, and Kaile Su. 2014. Improving Walk-SAT By Effective Tie-Breaking and Efficient Implementation. *Comput. J.* 58, 11 (11 2014), 2864–2875.

[9] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2015. On Parallel Scalable Uniform SAT Witness Generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 304–319.

[10] Jianfeng Chen, Vivek Nair, Rahul Krishna, and Tim Menzies. 2019. 'Sampling' as a Baseline Optimizer for Search-based Software Engineering. *IEEE Transactions on Software Engineering* 45, 6 (2019), 597–614.

[11] Paul Clements and Linda Northrop. 2001. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc. 467 pages.

[12] Carlos A. Coello Coello and Margarita Reyes Sierra. 2004. A Study of the Parallelization of a Coevolutionary Multi-objective Evolutionary Algorithm. In *Mexican International Conference on Artificial Intelligence (MICAI)*. Springer Berlin Heidelberg, 688–697.

[13] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2007. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA'07)*. Association for Computing Machinery, New York, NY, USA, 129–139.

[14] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-proving. *Communications of the ACM* 5, 5 (1962), 394–397.

[15] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[16] Stephane Doncieux, Alban Laflaquière, and Alexandre Coninx. 2019. Novelty Search: A Theoretical Perspective. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Prague, Czech Republic) *(GECCO'19)*. Association for Computing Machinery, New York, NY, USA, 99–106.

[17] Stephane Doncieux, Giuseppe Paolo, Alban Laflaquière, and Alexandre Coninx. 2020. Novelty Search makes Evolvability Inevitable. In *Proceedings of the Genetic and Evolutionary Computation Conference* (CancÃžn, Mexico) *(GECCO'20)*. Association for Computing Machinery, New York, NY, USA, 85–93.

[18] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient Sampling of SAT Solutions for Testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 549–559.

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

[20] Vibhav Gogate and Rina Dechter. 2006. A New Algorithm for Sampling CSP Solutions Uniformly at Random. In *Principles and Practice of Constraint Programming - CP 2006*, Frédéric Benhamou (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 711–715.

[21] Jorge Gomes, Pedro Mariano, and Anders Lyhne Christensen. 2015. Devising Effective Novelty Search Algorithms: A Comprehensive Empirical Study. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* (Madrid, Spain) *(GECCO'15)*. Association for Computing Machinery, New York, NY, USA, 943–950.

[22] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 301–311.

[23] Jianmei Guo, Jia Hui Liang, Kai Shi, Dingyu Yang, Jingsong Zhang, Krzysztof Czarnecki, Vijay Ganesh, and Huiqun Yu. 2019. SMTIBEA: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines. *Software & Systems Modeling* 18 (2019), 1447–1466.

[24] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines. In *The 37th International Conference on Software Engineering*, Vol. 1. 517–528.

[25] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Transactions on Software Engineering* 40, 7 (July 2014), 650–670.

[26] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. PLEDGE: A Product Line Editor and Test Generation Tool. In *Proceedings of the 17th International Software Product Line Conference Co-Located Workshops* (Tokyo, Japan) *(SPLC'13 Workshops)*. Association for Computing Machinery, New York, NY, USA, 126–129.

[27] Ruben Heradio, David Fernandez-Amoros, José A. Galindo, and David Benavides. 2020. Uniform and Scalable SAT-Sampling for Configurable Systems. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A* (Montreal, Quebec, Canada) *(SPLC '20)*. Association for Computing Machinery, New York, NY, USA, Article 17, 11 pages.

[28] Robert M. Hierons, Miqing Li, Xiaohui Liu, Sergio Segura, and Wei Zheng. 2016. SIP: Optimal Product Selection from Feature Models Using Many-Objective Evolutionary Optimization. *ACM Transactions on Software Engineering and Methodology* 25, 2, Article 17 (April 2016), 39 pages.

[29] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems* (Wellington, New Zealand) *(MODELS'11)*. Springer-Verlag, Berlin, Heidelberg, 638–652.

[30] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1* (Salvador, Brazil) *(SPLC'12)*. Association for Computing Machinery, New York, NY, USA, 46–55.

[31] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-Based Sampling of Software Configuration Spaces. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE'19)*. IEEE Press, 1084–1094.

[32] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. CMU/SEI-90-TR-21. SEI. Georgetown University.

[33] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: yet another sampling algorithm. In *VaMoS 20: 14th International Working Conference on Variability Modelling of Software-Intensive Systems, Magdeburg Germany, February 5-7, 2020*, Maxime Cordy, Mathieu Acher, Danilo Beuche, and Gunter Saake (Eds.). ACM, 4:1–4:10.

[34] Joel Lehman and O. Stanley Kenneth. 2011. Abandoning Objectives: Evolution Through the Search for Novelty Alone. *Evolutionary Computation* 19, 2 (2011), 189–223.

[35] Joel Lehman and Kenneth O. Stanley. 2008. Exploiting open-endedness to solve problems through the search for novelty. In *Proceedings of the Eleventh International Conference on Artificial Life (ALIFE XI)*. 329–336.

[36] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. SAT-based Analysis of Large Real-world Feature Models is Easy. In *Proceedings of the 19th International Conference on Software Product Line* (Nashville, Tennessee) *(SPLC '15)*. ACM, New York, NY, USA, 91–100.

[37] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 81–91.

[38] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 643–654.

[39] Jean Melo, Elvis Flesborg, Claus Brabrand, and Andrzej Wasowski. 2016. A Quantitative Analysis of Variability Warnings in Linux. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-Intensive Systems* (Salvador, Brazil) *(VaMoS'16)*. ACM, New York, NY, USA, 3–8.

[40] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 61–71.

[41] Jeho Oh, Paul Gazzillo, and Don Batory. 2019. T-Wise Coverage by Uniform Sampling. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC'19)*. Association for Computing Machinery, New York, NY, USA, 84–87.

[42] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Margaret Myers. 2019. *Uniform Sampling from Kconfig Feature Models*. Technical Report TR-19-02. The University of Texas at Austin.

[43] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Margaret Myers. 2020. *Scalable Uniform Sampling for Real-World Software Product Lines*. Technical Report TR-20-01. The University of Texas at Austin.

[44] Rafael Olaechea, Derek Rayside, Jianmei Guo, and Krzysztof Czarnecki. 2014. Comparison of exact and approximate multi-objective optimization for software product lines. In *The International Software Product Line Conference*. 92–101.

[45] Tobias Pett, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. *AutoSMP: An Evaluation Platform for Sampling Algorithms*. Association for Computing Machinery, New York, NY, USA, 41–44.

[46] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC'19)*. Association for Computing Machinery, New York, NY, USA, 78–83.

[47] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 240–251.

[48] Richard Pohl, Kim Lauenroth, and Klaus Pohl. 2011. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *IEEE/ACM International Conference on Automated Software Engineering*. 313–322.

[49] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE'15)*. IEEE Press, 342–352.

[50] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. 2013. Scalable product line configuration: A straw to break the camel's back. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 465–474.

[51] Mate Soos, Stephan Gocht, and Kuldeep S. Meel. 2020. Tinted, Detached, and Lazy CNF-XOR solving and its Applications to Counting and Sampling. In *Proceedings of International Conference on Computer-Aided Verification (CAV)*.

[52] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT Solvers on Industrial Feature Models. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems* (Magdeburg, Germany) *(VAMOS'20)*. Association for Computing Machinery, New York, NY, USA, Article 3, 9 pages.

[53] Marc Thurley. 2006. sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP. In *Theory and Applications of Satisfiability Testing - SAT 2006*, Armin Biere and Carla P. Gomes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 424–429.

[54] Yi Xiang, Han Huang, Miqing Li, Sizhe Li, and Xiaowei Yang. 2021. Looking For Novelty in Search-based Software Product Line Testing. *IEEE Transactions on Software Engineering* 1, 1 (2021), 1–1.

[55] Yi Xiang, Xiaowei Yang, Yuren Zhou, and Han Huang. 2020. Enhancing Decomposition-based Algorithms by Estimation of Distribution for Constrained Optimal Software Product Selection. *IEEE Transactions on Evolutionary Computation* 24, 2 (2020), 245–259.

[56] Yi Xiang, Xiaowei Yang, Yuren Zhou, Zibin Zheng, Miqing Li, and Han Huang. 2020. Going deeper with optimal software products selection using many-objective optimization and satisfiability solvers. *Empirical Software Engineering* 25 (2020), 591–626.

[57] Yi Xiang, Yuren Zhou, Zibin Zheng, and Miqing Li. 2018. Configuring Software Product Lines by Combining Many-Objective Optimization and SAT Solvers. *ACM Transactions on Software Engineering and Methodology* 26, 4, Article 14 (Feb. 2018), 46 pages.