Random Search versus Uniform Sampling
A counter-intuitive result

Mathieu Vavrille

Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004, F-44000 Nantes, France
mathieu.vavrille@univ-nantes.fr

Keywords : *Constraint Programming, Solution Coverage, Software Testing, Software Product Line, Feature Model*

1 Introduction

With the increase in software size and complexity comes the need of good automatically generated test suites. For example, the Linux kernel has thousands of features (e.g. options, libraries, ...). A commonly used quality measure of a test suite is the t-wise coverage. It aims at analysing how many interactions of t features are tested.

Multiple tools already exist for this task (such as [2]) and can guarantee that all the possibles combinations of t features are present in the test suite. However, these tools often require computing beforehand the set of all t-wise combinations. This set can be as big as $\binom{|\mathcal{F}|}{t}2^t$ (where $\mathcal{F}$ is the set of features). On software of thousands of features it is intractable to try to enumerate all the t-wise combinations when t is growing.

With this consideration in mind, more recent approaches (such as [1]) use the improvements done in SAT samplers to generate a test suite. There is no need to generate the set of all t-wise combinations, at the cost of the coverage guarantee. Yet the experiments have shown that the coverage is good, and these approaches can generate solutions on demand.

We show here that a random search, i.e. a search strategy that chooses randomly a feature, and add it to a configuration or not, outperforms a uniform sampler for the task of t-wise coverage on feature models. This result is counter-intuitive because a random search is a biased sampler, compared to a uniform sampler whose behaviour is proved.

2 Problem Description

A *Feature Model* is a compact representation of all the possible configurations of a Software Product Line, on a set $\mathcal{F}$ of features. An example of feature model of a game engine is given in Figure 1: a game has a `style`, and optionally a `Multiplayer` mode; the style is either `Shooter` or `Racing`, and the `Multiplayer` mode can be `Local` or `Online`, or both. Through a tree structure and propositional formulas, feature models define the set of allowed configurations $\mathcal{S}$, where a configuration is a subset of $\mathcal{F}$.

A t-wise combination is a mapping $\sigma : \mathcal{F}' \to \{0, 1\}$ with $\mathcal{F}' \subseteq \mathcal{F}$ and $|\mathcal{F}'| = t$. A configuration $C \in \mathcal{S}$ is said to cover a combination σ if for all $f \in \mathcal{F}$, $\sigma(f) = 0 \Leftrightarrow f \in C$. For example on the `Game` feature model, a developer would like to test if the 2-wise (pairwise) combination $\{\texttt{Racing} \mapsto 1, \texttt{Multiplayer} \mapsto 0\}$ (i.e. a single player racing game) can be generated by the game engine.

Given two integers t and k, we are interested in the problem of finding k configurations that cover the most t-wise combinations possible. In the following section we apply two approaches (random search and uniform sampling) to generate test suites and compare the number of t-wise combinations covered. These two approaches have the advantage of not having to generate the set of all the combinations (there are possibly $\binom{n}{t}2^t$ t-wise combinations on n features).

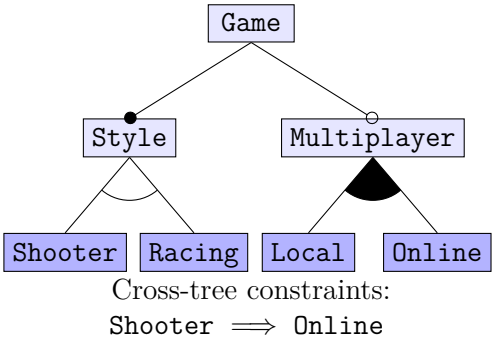

FIG. 1: Example of Feature Model

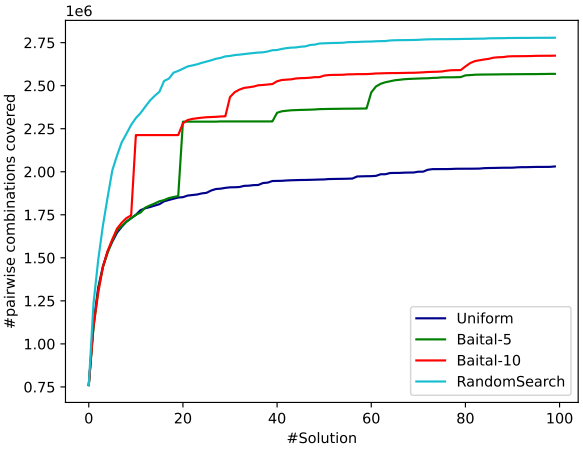

FIG. 2: Evolution of the pairwise coverage on the instance `brutus`

3 Experiments

We apply the random search strategy in a constraint solver to generate solutions of feature models. During the decision phase of a constraint solver, a variable X and a value v have to be chosen to continue the solving. The strategy RandomSearch chooses this variable and value at random (and uniformly among all possibilities). Using this strategy to generate solution makes some solutions more likely to be selected than others. We used `choco-solver` to implement RandomSearch.

We compare it to uniform sampling, and to the state-of-the-art tool Baital [1]. Baital is a tool generating a test suite based on a weighted sampler. It first samples uniformly m configurations. Then, knowing these m configurations, weights are chosen to make it more likely to sample unseen combinations. By applying many rounds of this procedure a test suite is generated. In the experiments, we tested Baital with 5 and 10 rounds, and searched for 100 configurations.

We used a benchmark of feature-models from different origins in UVL format.[1] An example of evolution of the number of 2-wise combinations found is given in Figure 2. It shows that RandomSearch outperforms the uniform sampler, and even Baital. On average (using the geometric average), RandomSearch achieves the same coverage as Baital-10 with three times fewer solutions. Due to its simplicity, on average, RandomSeach is 100 times faster than Baital-10. These experiments show that a simpler random process can lead to very good coverage.

References

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

[2] Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. Greedy combinatorial test case generation using unsatisfiable cores. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 614–624. ACM, 2016.

[1] `https://github.com/Universal-Variability-Language/uvl-models`