GenGen – Input Generation with Genetic Algorithm for Performance Testing

Anuar Talipov *Google, Seoul* Shynar Torekhan *MindsLab, Seoul* Azamat Smagulov *Yelp, Hamburg* Azret Kenzhaliev *Yelp, London*

Performance testing is of critical importance due to the emergence of scalable high performance, high load, and safety critical systems. Here, we propose a model based on a genetic algorithm for generating inputs, which can adequately test functions performance for programs written in Python programming language. We provide an extendable tool with many useful features for developers. **Repository of the project** is available here: `https://github.com/AnuarTB/gengen`

I. Introduction

Load tests aim to validate whether a system performance is acceptable under peak conditions. Load testing is crucial nowadays, especially in the scalable and safety-critical systems. Load testing can assist in the detection of performance anomalies and there are many tools for load testing. Most of those tools treat software as a black box and focus on increasing load by providing larger input sizes and ignoring the particular input values. However, only increasing the size of the input can be inefficient.

First, some functions' performance highly depends on the values inside the input and not on the size/length of the input. Consequently, random choice of values can lead to underestimating functions response time by missing an opportunity to detect a performance fault.

Second, increasing the input size can be expensive for loading the system, if increasing the input size requires additional expensive resources in order execute the test (ex: additional disk space, bandwidth).

Third, only increasing the input size can cause the system to perform the computation of the same nature due to bad diversification of the input contents.

In this project, we wanted to address these problems. Yet, identifying such inputs can be extremely challenging because it requires understanding of the program internals. To address this challenge, we use the **genetic algorithm** for the selection of the best combination of input values that can deliver an equivalent or bigger load with the equivalent or smaller input sizes, which in return can reduce testing infrastructure requirements and can provide a more accurate characterization of scenarios where a system behaves poorly.

II. Methodology

In this section, we will consider how we have employed genetic algorithm to generate the input with high runtime

A. Background: Genetic Algorithm

The process begins with a set of individuals, which is called a **population**. An individual is characterized by a set of parameters (variables) known as **genes**.

A **fitness function** determines how fit an individual is, the ability to compete with other individuals to be reproduced later.

Selection is a process where individuals (inputs) are chosen to breed and produce a new generation [of inputs]. The parents undergo **crossover** with each other to pass stronger genes (input parts) to the offspring.

Among the new offspring, some portion of it with some small probability might be subject to the **mutation** - process where some of the individuals genes are randomly altered.

B. Baseline

For our problem the obvious choice of the fitness function was $T_{x,i}$ the run-time of the function f under the input i. At first, we have generated the random population P whose size $|P|$ is chosen by user (default: 100). After that the algorithm will have K iterations, trying to improve the population. At each iteration, for each new individual, the genetic algorithm will select two parents $parent_1$ and $parent_2$ for breeding, following fitness proportionate selection [1]. The new individual I will then have a chance of mutation, in order to diversify the pool of inputs. The population at the end of iteration is mixed with the new one (default: 20/80%), and some of the weak individuals are discarded.

C. Challenges

Although the random generation of the inputs is an easy task, combining them for crossover and mutation for producing adversarial input is a difficult ad-hoc problem. The problem with our mutation and crossover strategies observed during the evaluation of our tool was the fact that there is no universal strategy that would produce adversarial inputs for problems of every type. To elaborate, for example, we can not have the same input type for our Hash Map testing and Insertion Sort testing. For the former, worst case is an array containing the same numbers, and for the latter, the worst case is a decreasing array of numbers. There is a low probability that one strategy would perform well in both cases. Thus, we have implemented several strategies for crossover and mutation that user can choose among for generating input, which will be best fit for the problem.

Another aspect of GenGen is that our model has many hyper-parameters. The user can tune hyper-parameters by oneself, however, the problem is that some hyper-parameter configuration can perform significantly differ-

ent than the other. That's why during evaluation we have set almost all hyper-parameters to the same values, with the exception of some, which are dependent on the task (ex: different functions need to receive different types of parameters(List, Int, etc)) and, therefore, couldn't be set to the same values.

D. Logging Feature

Nowadays, processing power of computers is very high. Because of that, developers may not notice that their commit introduced performance fault. Therefore, we provide a logging feature inside our tool. It can visualize the history of system's performance, so that developers can trace whether the commit introduced performance fault and decreased performance of the system.

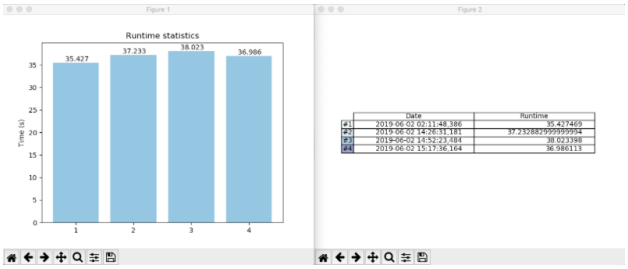

FIG. 1. Logging Feature

III. Results and Evaluation

A. Experiment setup

For the experiment, we have set up a machine with the following characteristics:
CPU: Intel Core i5-7300HQ CPU @ 2.50GHz × 4
OS: Ubuntu 18.04.2 LTS
RAM: 8 GB
Python version: 3.7.3

B. Evaluation method

We used an evaluation method widely used in the papers on test-input generation. We compared GenGen's generated input run-times against run-times of random and theoretically worst inputs. For this, we had to find algorithm whose performance highly depends on the values inside the input and not the size/length of the input.

We have implemented some basic algorithms that are listed in Table II. These are algorithms with known worst run-time cases, so we could easily construct the worst cases and compare our results with them.

For Hash Map testing, we have considered an (int, int) hash table with chaining, where simple modulo operation was used as a hash function. The theoretical worst in this case is an array of the same elements, which consequently have the same hash function value and are mapped in the same chain inside a hash table.

For insertion sort, the worst theoretical input is a decreasing array, on which the algorithm performs in $O(n^2)$ running time.

For BST, its basic implementation, we inserted a sequence of numbers and searched for all them one-by-one. The worst case for BST is a sorted sequence of numbers, which causes BST to have a height of n, where n is the number of elements in the sequence.

C. Results

No.	Random	Theor. Worst	GenGen
Hash Map	0.055393	3.659568	2.937697
Insert. Sort	1.095497	2.068523	1.400730
Bubble Sort	2.668995	3.368677	3.258314
BST	0.040536	5.098057	3.134461

TABLE I. Runtime comparisons

No.	Random	GenGen	Theor. Worst
Hash Map	1.88%	80.27%	100%
Insert. Sort	52.96%	67.72%	100%
Bubble Sort	79.23%	96.72%	100%
BST	0.79%	61.48%	100%

TABLE II. Comparison of GenGen input's runtime against Random and Theoretical worst input

We measured runtime in seconds by using Python's *time.process_time*(). According to Python documentation, this method of *time* library can be used for benchmarking purposes. The runtime measurements can be viewed in Table I. In Table II, the comparison of GenGen inputs' runtime against random and theoretically worst inputs' runtimes can be viewed. Theoretical worst input's runtime is considered to be 100%, i.e. the upper bound of the runtime of the program under test. According to the comparison, GenGen's performance is quite close to the theoretically worst input's performance and considerably outperforms random input's performance.

[1] Fitness proportionate selection
[2] BST implementation