Practical Property-based Testing in Scientific Software Proposal

ECS260

October 22nd 2015

I. Team members

- Mousumi Chattaraj
- Luiz Irber
- Li Li

II. Problem Description

Correct software is expensive to develop. Scientific software suffers even more due to lack of software development expertise among scientists and maintainability problems: people leave labs, usually without creating enough documentation or processes for continued development. We would like to explore ways to increase our confidence in the correctness of scientific software or, more realistically, find cheap ways of show when it is incorrect.

The khmer project is a mixed Python/C++ codebase implementing a few core data structures (sketches) and many methods and analysis on top of it. The source code is available on GitHub under a 3-clause BSD license, and every pull request is tested on a Continuous Integration server (Jenkins). It has high coverage (90% line coverage) derived from manually written (~600) unit tests, but many tests were written just to trigger coverage, with no particular focus on correctness. Can we do better, without increasing development cost (too much)?

III. Technical Approach

Property-based testing has its roots on the Haskell programming language, in a module called QuickCheck. Instead of defining test cases for specific values (like corner cases), QuickCheck proposes that tests should be verifiable by checking properties and invariants that describe what is expected for any input, and then generating inputs to try to falsify the test.

Haskell is a functional language that avoids side-effects ("pure"), making it simpler to reason about properties as mathematical aspects, and apply concepts across many domains. This approach doesn't work so well in other languages, and for Python there is a library called Hypothesis implementing a mix of unit and property-based testing, and fuzzing.

Our plan is to understand what are the best options to generate meaningful Hypothesis tests to find incorrect behavior in khmer. To achieve that we plan to do progressive coverage and exploration of the khmer codebase, starting from simpler pieces and building up knowledge to be able to tackle complex regions. We split the plan in four areas:

1) Hashing, the implementation of hash functions used to map strings to an integer space (and in some cases the reverse operation too).
2) Hyperloglog, an algorithm that approximates the number of distinct elements in a multiset. It is a probabilistic cardinality estimator that uses significantly less (and constant) memory to obtain an approximation of the cardinality.
3) Bloom Filters, a data structure designed to check the presence of an element in a set. False positive matches are possible, but false negatives are not. A bit vector (Figure 1) is the base structure of a bloom filter, where each empty cell represents a bit and the number below is its index. In order to add an element to the table, we will hash it a few times and then set the bits in the bit vector at the index of those hashes to 1.

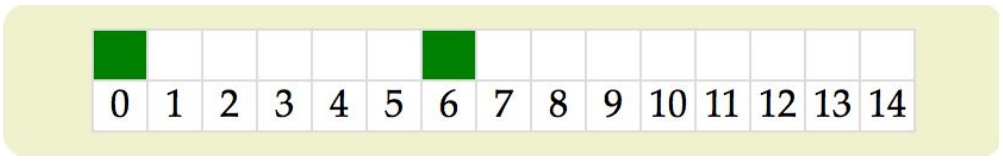

Figure 1: Example of a simple bit vector.

4) Count-Min Sketch, a probabilistic data structure that serves as a frequency table of events in a stream of data by using hash functions to map events to frequencies. The goal is to consume a stream of event one at a time, and count the frequency of different types of events in it without storing the whole stream.

IV. Evaluation Methodology

It is hard to find a good metric for success in this project: number of bugs found wouldn't necessarily represent interesting or important bugs, but at the same time we can't guarantee we'll find any non-trivial bug. There is also the complexity of reasoning about the data structures to find testable properties and how to set up the tests. Nonetheless, we would like to track how many method calls we can cover, starting from the core data structures and moving to specialized methods (the ones used directly by data analysis scripts). We're not aiming at complete code coverage, but it is likely that we can get good coverage with the Hypothesis tests, even if inconsistent between runs because new test cases will be generated.

Finally, we want to document and generates guidelines for testing similar software, be it other biological data analysis projects or scientific in general. This document can be used as a quickstart and introduction to these testing techniques in specialized fields.

V. List of Preliminary Related Work

- Claessen, Koen, and John Hughes. "QuickCheck." *ACM SIGPLAN Notices SIGPLAN Not*. 46.4 (2011): 53. Web. (https://dl.acm.org/citation.cfm?id=1988046)
- Crusoe MR, Alameldin HF, Awad S *et al.* "The khmer software package: enabling efficient nucleotide sequence analysis [version 1; referees: 2 approved, 1 approved with reservations]" *F1000Research* 2015, **4**:900 (doi: 10.12688/f1000research.6924.1)
- MacIver, David R. "Welcome to Hypothesis!" *Hypothesis*. N.p., n.d. Web. 22 Oct. 2015. (https://hypothesis.readthedocs.org/en/latest/)
- Mill, Bill. "Bloom Filters by Example." *Bloom Filters by Example*. N.p., n.d. Web. 22 Oct. 2015. (http://billmill.org/bloomfilter-tutorial/)
- MacIver, David R. "The Economics of Software Correctness." *David R. MacIver*. N.p., 5 Oct. 2015. Web. 22 Oct. 2015. (http://www.drmaciver.com/2015/10/the-economics-of-software-correctness/)