Milestone 1

Mousumi Chattaraj, Luiz Irber, and Li Li

University of California, Davis
{mchattaraj,lcirberjr,llili}@ucdavis.edu

1. INTRODUCTION

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)?

2. MOTIVATING EXAMPLE

Hashing is an essential operation in khmer, since all the core data structures depend on good hash functions to perform correctly. Since the development process is public and coordinated through GitHub, an exploratory branch was created, with an accompanying issue and discussion: https://github.com/dib-lab/khmer/issues/990

This original exploration used a property from the hash function, reversibility: any string used as input generates a hash value (an integer), which can be used to generate the original string again. The implementation using Hypothesis is the following:

```
@given(text("ACGT"))
def test_forward_hash_no_rc(kmer):
    ksize = len(kmer)
    assume(ksize > 0)
    assume(ksize < 32)

    rh = reverse_hash(
        forward_hash_no_rc(kmer, ksize),
        ksize)
    assert rh == kmer
```

This example contains some important aspects:

1. the input *kmer* is generated automatically from a specification (the *given* decorator), meaning each *kmer* is a string with alphabet "ACGT".
2. the *assume* function asserts that the generated *kmer* is not an empty string, and also not longer than 32 characters.
3. the conditions are tested using normal Python syntax (assert, variables, function calls)

The input specification was added later, by the way. The code was not prepared to deal with empty strings, strings longer than 32 characters or with a different alphabet. These are all bugs that need to be fixed, and they are good to show the approach. This example is very similar to typical unit testing in Python, while the Hypothesis extensions to the concept allow both property-based testing and fuzzing usage.

3. TECHNICAL APPROACHES

3.1 Incremental exploration

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 into four areas:

3.1.1 Hashing

Hashing is the implementation of hash functions used to map strings to an integer space, and in some cases this operation is reversed. It maps data of arbitrary size to data of fixed size.

3.1.2 HyperLogLog

HyperLogLog is 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. In the algorithm a hash function is applied to each element in the original multiset to obtain uniformly-distributed random numbers with the same cardinality as the original. Estimating the cardinality of a multiset by calculating the maximum number of leading zeros in the binary representation of each number is the basis of HyperLogLog algorithm.

3.1.3 Bloom Filter

A Bloom Filter is a data structure designed to check the presence of an element in a set rapidly and memory-efficiently. A bit vector 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. False positive matches are possible, but false negatives are not.

3.1.4 Count-Min Sketch

Count-Min Sketch is 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.

4. RELATED WORK

4.1 QuickCheck

Property-based testing has its roots on the Haskell programming language, in a module called QuickCheck (Claessen and Hughes [1]). 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 testing, property-based testing, and fuzzing.

4.2 Hypothesis

Hypothesis (MacIver [4]) is a Python library for creating simple and powerful unit tests to find edge cases. It asks user to write cases that assert something should be true for every case instead of just the ones that could be thought of. For a normal unit test, the first step is to setup some data, and then performing some operations on the data, and at last asserting something about the result. However Hypothesis asks user to write test cases for all data matching some specification, and then performs some operation on the data, and makes the assertion in the end. This is often called property based testing. Hypothesis works by generating random data that match the specification for the data and checks if the guarantee still holds in the case. If not it will take the example and cut its size to simplify it until a much smaller example that still causes the problem is found. Hypothesis then saves that example for later use.

4.3 khmer

khmer (Crusoe et al. [2]) is a free software library that works with fixed length DNA words (also known as k-mers) and implements a probabilistic k-mer counting data structure. Currently khmer is implemented in $C++$ and Python. The data structures and graph traversal code are implemented in $C++$, and then wrapped for Python in hand-written C code. khmer is primarily developed on Linux for Python 2.7 and 64-bit processors, and several core developers use Mac OS X. k-mers are a common abstraction in DNA sequence analysis that enable alignment-free sequence analysis and comparison. However the dramatic increase in sequence data generation has led to the development of data structures ad algorithms to discover possible improvements to k-mer-based approaches. The khmer project provides several efficient data structures and algorithms to analyze short-read nucleotide sequencing data while emphasizing online analysis, low memory data structures and streaming algorithms.

4.4 The economics of software correctness

According to MacIver [3], it is almost impossible for someone to write a significant piece of correct software. It is reasonable to write bug-free small libraries, but the chances to write a whole perfect program are almost zero. The problem is not that people do not know how to write correct software, it is that correct software is too expensive. Being too expensive means that nobody would be willing to pay a price for the software, or a competing product could be released much earlier. The result of this is shipping buggy software to users. The problem is: bugs found by users are more expensive than bugs found before a user sees them, since in the former case the result could be lost users, lost time and theft. Hypothesis is a good example of reducing the effort of finding bugs.

5. 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.

6. TEAM MEMBER CONTRIBUTIONS

Mousumi Chattaraj: Evaluation methodology, milestone 2 tasks.

Luiz Irber: Motivating example, Introduction, milestone 2 tasks.

Li Li: Technical approaches, Related work, milestone 2 tasks.

7. MILESTONE 2 TASKS

We'll use test oracles to check basic sketch functionality. Each of the core data structures have exact representation equivalents, which can be used to verify some properties of the probabilistic ones:

1. Bloom filters: implement the set membership operation, can be tested by setting up a *set* in Python and adding all the generated inputs to both. We can assert that every element present in the set is also present in the Bloom Filter.

2. Count-min sketch: implements frequency counting. Can be teste by setting up a *collections.Counter* object, and checking that every element in the counter is also available with equal or higher frequency on the Count-min sketch.

3. HyperLogLog: implements set cardinality estimation. The setup is similar to Bloom Filter, but we check for the set size instead of membership of every element.

These tasks also allow us to start exploring all the areas we described in the technical approach, and helps to describe all the main areas of code to all the project members.

8. REFERENCES

[1] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011. URL `http://dl.acm.org/citation.cfm?id=1988046`.

[2] M. R. Crusoe, H. F. Alameldin, S. Awad, E. Boucher, A. Caldwell, R. Cartwright, A. Charbonneau, B. Constantinides, G. Edvenson, S. Fay, J. Fenton, T. Fenzl, J. Fish, L. Garcia-Gutierrez, P. Garland, J. Gluck, I. GonzÃąlez, S. Guermond, J. Guo, A. Gupta, J. R. Herr, A. Howe, A. Hyer, A. HÃd'rpfer, L. Irber, R. Kidd, D. Lin, J. Lippi, T. Mansour, P. McA'Nulty, E. McDonald, J. Mizzi, K. D. Murray, J. R. Nahum, K. Nanlohy, A. J. Nederbragt, H. Ortiz-Zuazaga, J. Ory, J. Pell, C. Pepe-Ranney, Z. N. Russ, E. Schwarz, C. Scott, J. Seaman, S. Sievert, J. Simpson, C. T. Skennerton, J. Spencer, R. Srinivasan, D. Standage, J. A. Stapleton, S. R. Steinman, J. Stein, B. Taylor, W. Trimble, H. L. Wiencko, M. Wright, B. Wyss, Q. Zhang, e. zyme, and C. T. Brown. The khmer software package: enabling efficient nucleotide sequence analysis. *F1000Research*, Sept. 2015. doi: 10.12688/f1000research.6924.1. URL `http://f1000research.com/articles/4-900/v1`.

[3] D. R. MacIver. The economics of software correctness, May 2015. URL `http://www.drmaciver.com/2015/10/the-economics-of-software-correctness/`.

[4] D. R. MacIver. Welcome to Hypothesis! âĂŤ Hypothesis 1.14.0 documentation, Nov. 2015. URL `https://hypothesis.readthedocs.org/en/latest/`.