Diversity in Generator Output

Introduction

Unit testing is the dominant approach to testing in the software industry. Unit testing requires the programmer explicitly defines inputs and expected outputs for the function under test. Therefore, the effectiveness of unit testing is limited by the amount of effort the programmer expends in creating distinct input and output pairs.

Property based testing, or generative testing, is a lightweight extension of unit testing that aims to improve on unit testing. Instead of the programmer explicitly specifying input and output pairs, the programmer defines generators and properties. A **generator** is a program for generating test inputs, and **properties** are conditions that must hold for the output. By using the computer to generate the input, many more inputs can be generated than when the programmer specifies them by hand and inputs that the programmer may have overlooked can be created.

Property based testing has increasing adoption within industry, with libraries available in many popular languages. However, this doesn't mean existing practice cannot be improved. Our goal here is to make such an improvement by focusing on how generators create the test inputs.

Property based testing libraries provides an API to construct generators. The standard approach is an implementation of the probability monad, which is simple to implement and to use. The generators so constructed can be viewed as functions from a source of randomness to an output of the desired type, which is the input to the function under test.

Ideally, the output of a generator should be highly **diverse**. Intuitively we want the generated outputs to be as different from each other as possible, to maximize the chances of finding bugs. (We'll define this more formally later.) Random generation is not the best choice for achieving this goal for two reasons:

1. A generator defines a distribution over test inputs. Most programmers are rarely statisticians. In practice little attention is paid to the properties of this distribution and generators often have undesirable properties.
2. Random generation is an inefficient way to explore the space of test inputs. Randomly generated data often displays "clumping": data points that are near to one another, or even exactly the same.

Our focus here is on the second point, though we'll briefly touch on the first as well.

In this paper we look at the task of generating diverse inputs in property based testing. We start by illustrating the problem in practice. When then formalize the setting, and develop a toolbox for constructing algorithms that generate

input. We then discuss algorithms to increase diversity, and test these algorithms on several real-world problems.

The Problem of Diversity in Test Input Generation

Let's start with an example of a very simple property based test, taken from the Doodle library. Doodle, written in Scala, is a library for two-dimensional graphics. Color is a core abstraction in Doodle, and colors can be specified in two equivalent ways:

- as a triple of red, green, and blue values (RGB); or
- as a triple of hue, saturation, and lightness (HSL).

The HSL representation is easier for humans to work with, while RGB is what the computer uses to display color. Conversion between the two formats is therefore necessary, but this is not straightforward. The RGB representation defines a cube, with each component an unsigned byte. The HSL representation defines a cylinder, as hue is represented as an angle while saturation and lightness are both floating point numbers in the range 0.0 to 1.0. Therefore, the conversion requires a relatively involved algorithm.

There is a simple property that checks the conversion: converting an HSL color to RGB and back should yield, approximately, the same color. In other words, the conversion from HSL to RGB should be invertible. We can also check the conversion going the other way: from RGB to HSL to RGB. Below is a test for both these properties, encoded in the ScalaCheck property-testing library.

```
object ColorSpec extends Properties("Color properties") {
  import doodle.arbitrary._
  import Color._

  property(".toRGBA andThen .toHSLA is the identity") =
    forAll { (hsla: HSLA) =>
      (hsla ~= (hsla.toRGBA.toHSLA))
    }

  property(".toHSLA andThen .toRGBA is the identity") =
    forAll { (rgba: RGBA) =>
      (rgba ~= (rgba.toHSLA.toRGBA))
    }
}
```

The `~=` operator denotes approximate equality.

These tests rely on generators to produce HSL and RGB colors. The definitions of the HSL generator are shown below. In words, this simply samples an angle uniformly in the range 0 to 360 degrees, and saturation and lightness uniformly in the range 0.0 to 1.0.

```
val angle: Gen[Angle] =
  Gen.choose(0, 360).map(_.degrees)

val normalized: Gen[Normalized] =
  Gen.choose(0.0, 1.0).map(_.normalized)

val color: Gen[HSLA] =
  for {
    h <- angle
    s <- normalized
    l <- normalized
    a <- normalized
  } yield Color.HSLA(h, s, l, a)
```

There is a problem with this generator. Fig. 1 shows 100 and 1000 colors respectively, sampled as described above except that lightness is fixed to 0.7. Fixing lightness makes no substantive change to the points we wish to illustrate, but the two-dimensional visualization is easier to interpret.

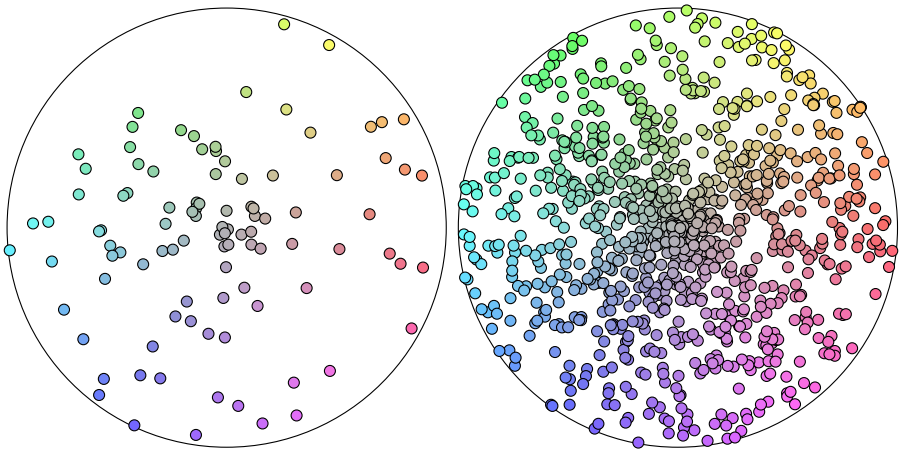

Figure 1: 100 and 1000 random colors, keeping lightness constant but allowing hue and saturation to vary

This illustrates both issues we originally raised with random sampling: the straightforward definitions of generators may have undesirable properties, and that random sampling is inefficient. Let's look at both in turn.

Firstly, notice that with 1000 samples the density of samples is greater at the centre of the circle than the edges. This is a flaw with our definition of the generating distribution. Chosing angle (hue) and radius (saturation) both uniformly at random does not give a uniform distribution over a circle because the area of the circle is more concentrated to the outside of the circle. To

correctly sample uniformly over the circle we should bias to larger values of the radius; to be precise, if we transform that uniform distribution by the square root we get the desired output. This is well known in certain circles (pun very much intended) but we should not expect the average programmer to be aware of this.

Looking at the picture of 100 samples notice how the samples are clumped together, both because of the issue with the definition of the generator discussed above and also just because of the nature of random samples. This clumping is also evident in the picture showing 1000 samples. Without some prior knowledge of where in the sample space bugs are found we should aim to sample with uniform density across the space of test inputs.

This simple example illustrates the issues with random sampling. Most programmer deal with discrete structures, like lists and trees, which do not lend themselves to such simple visualization and analysis. What we desire is a general purpose algorithm that can take a description of a generator and produce high quality output. As our first step to this, in the next section we formalize the problem of test input generation as one of graph search, and introduce a toolbox of methods for constructing graph search algorithms.

Test Input Generation as Graph Search

We're now ready to formalize our setting. We start by formalizing the testing setting, and then turn to generators.

The function under test is an arbitrary function $f : A \to B$. Our only requirement is that f is a pure function; in other words, it is deterministic. (If this is not the case the whole testing enterprise is pointless.)

A generator is a function $g : \mathit{Random} \to A$, where *Random* is some source of randomness. Our generators follow the structure in "Parsing Randomness".

Finally we have some post-condition $\mathit{post} : B \to \mathit{Bool}$, that determines if f's output is acceptable.

Generators define a directed graph, or more specifically a tree, of possible program executions. Vertices in the graph are calls to `Select`, and edges are the deterministic execution that occurs between one call to `Select` and another. This allows us to formulate the data generation problem as one of graph search. Generating data involves following edges through the graph until they terminate at a node with no further edges: a **sink**. How we choose between available edges is where things get interesting. In the next section we discuss algorithms for graph search.

This needs more exposition

Formulating generators as directed graphs immediately suggests classical algorithms for graph search: breadth-first search, depth-first search, and a^*-search.

There is also the current practice of random generation.

We can describe some of the characteristics that differentiate algorithms:

- Deterministic vs random: deterministic algorithms always generate output in the same order, whereas random algorithms do not. Breadth-first and depth-first search are exampels of deterministic algorithms.
- Adaptive vs non-adaptive. Adaptive algorithms take account of information gained in prior runs to attempt to guide the search. We have not yet seen any adaptive algorithms. The No Free Lunch theorem suggests adaptive algorithms cannot out perform non-adaptive algorithms over all possible generators, but we are only concerning ourselves with the relatively small subset of actual generators used in practice. In this situation we might expect an adaptive algorithm to out-perform a non-adaptive one.
- Memory usage. - In general the size of the set of test inputs is infinite, so any algorithm with memory requirements proportional to this size may be problematic. Random sampling, which is the current practice, requires no memory.

We now describe tools we can use to build more complex algorithms.

Quasi-random Sequences

Quasi-random sequences are deterministic algorithms that generate output with similar properties to random sequences. There are two classes of algorithms of interest to us:

- low discrepancy sequences, such as the Halton and Sobol sequence, which are the quasi-random equivalent of the uniform distribution on $\mathbb{R}^N$; and
- quasi-random walks, such as the rotor-router algorithm, which are the quasi-random equivalent of a random walk on a graph.

Bayesian Surprise

One way to measure diversity is to use Bayesian surprise. Given data D, in this case samples from a generator, and a model M, in this case a generator, Bayesian surprise is defined as

$$Surprise(D, M) = KL(P(M|D), P(M)) \tag{1}$$

KL is the KL-divergence, defined as

$$KL(P(M|D), P(M)) = \int_M P(M|D) log\big(\frac{P(M|D)}{P(M)}\big) dM \tag{2}$$

There are several terms in the equations above that are standard in Bayesian statistics but it is perhaps not clear how they relate to generators. What follows is brief sketch of Bayesian statistics.

Let's assume we have some way of generating data, our generator, which assigns a probability to each possible output. In other words, the generator defines a probability distribution. In keeping with the definitions above, we'll label our generator M, and the data D. The generator defines $P(D|M)$, the probability of the data given the generator.

In the Bayesian view we don't commit to single generator but instead consider a distribution over generators $P(M)$. This is known as the **prior distribution**. When we see data we can update this distribution to produce $P(M|D)$, the distribution over generators given data, known as the **posterior distribution**. How do we compute the posterior? Bayes theorem tells us that

$$P(M|D) = \frac{P(D|M)P(M)}{P(D)} \tag{3}$$

where

$$P(D) = \int_M P(D|M)P(M) \tag{4}$$

This tells us in theory how we can compute the posterior, but how does it work in practice? The answer is that it depends on the particular form of the prior. There is a class of prior distributions for which we easily compute the posterior. These are known as **conjugate priors**. If M is choosing between discrete choices (a categorical distribution) and we use the Dirichlet distribution as our prior, the posterior will also be a Dirichlet distribution. There is also a simple rule that relates the posterior to the prior. Hence the Dirichlet is called the conjugate prior for the categorical distribution.

Note the `Select` function in **Parsing Randomness** is a categorical distribution, and thus the Dirichlet is an appropriate prior for it.

We've now defined all the terms in the definition of Bayesian surprise, but it's worth quickly emphasizing the implications for generators. The first implication is that we don't view a generator as defining a particular distribution, but rather defining a family of distributions. This is actually implicit in the definition of free generators, as they do not have probabilities attached to choices. Developers do not pay any particular attention to the distribution their generators define, in my experience, so this also does not conflict with the practice of property based testing. The second implication is that the posterior makes the observed data more likely, and hence less surprising should it be generated again, and therefore the posterior acts as a summary of the data that has been generated to date.

Markov Decision Processes

A Markov Decision Process (MDP)

In our case the MDP is deterministic, and we are purely exploration.

Algorithms for Graph Search

In this section we introduce a toolbox for building graph search algorithms.