Automated Test Input Generation Strategies

Kyle Dewey
Prepared for COMP 587

1 Summary and Background

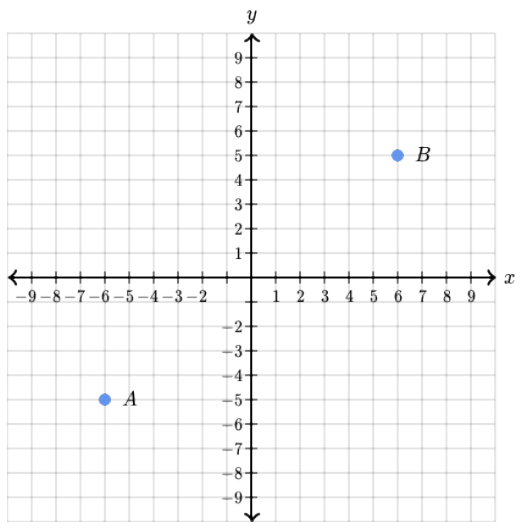
These notes detail some of the discussion we had in class on Monday, October 15, 2018. These notes introduce *random* and *bounded-exhaustive* testing.

1.1 Starting Specific: `min`

To help explain the concepts in these notes, let's start with a concrete example. Say we want to test the following `min` method, written in Java:

```
public static int min(int x, int y) {
  if (x < y) {
    return x;
  } else {
    return y;
  }
}
```

As shown, the inputs to `min` are the two `int` values `x` and `y`. Different `x`, `y` values form different *test inputs*, where a single test input consists of a single value for `x`, along with a single value for `y`. Specific to `min`, we can visualize different possible test inputs as individual points on a 2D coordinate plane, like so:



This particular example shows two specific test inputs, namely `-6,-5` (A) and `6,5` (B).

If we wanted to talk about *all* possible test inputs for `min`, we would have a point at each gridline intersection.

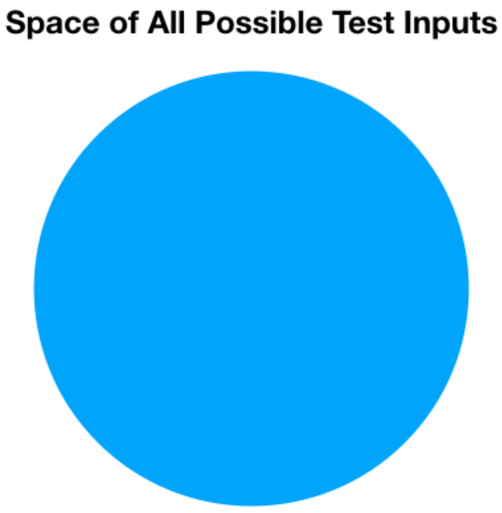
1.2 Generalizing Space of Test Inputs

Instead of `min`, let's say we wanted to test the following Java method:

```
public static int multThree(int a, int b, int c) {
  return a * b * c;
}
```

Each test input to `multThree` would consist of an `a`, `b`, and `c` triplet. If we were to map out the space as we did before, this would form a 3D space, with one dimension per variable. In general, each input variable forms one dimension in the space of test inputs, so a function with N inputs would need an N-dimensional space to model every possible input.

Because these spaces quickly get very complex, we need some restrictions for the rest of this discussion to help keep everything sane. Let's assume that the space of *all* test inputs is representable by a circle, shown below:



This circle encompasses all possible test inputs to whatever it is we are testing. Even for relatively simple things (e.g., `min` and `multThree` above), this circle will often contain **trillions** of test inputs. For example, the space of test inputs to `min` contains `MAX_INT * MAX_INT` elements ($(2^{31} - 1)^2$) For most realistic problems, this space is literally infinite (e.g., the user can enter any unbounded string, the programmer may write any program when testing a compiler, etc.).

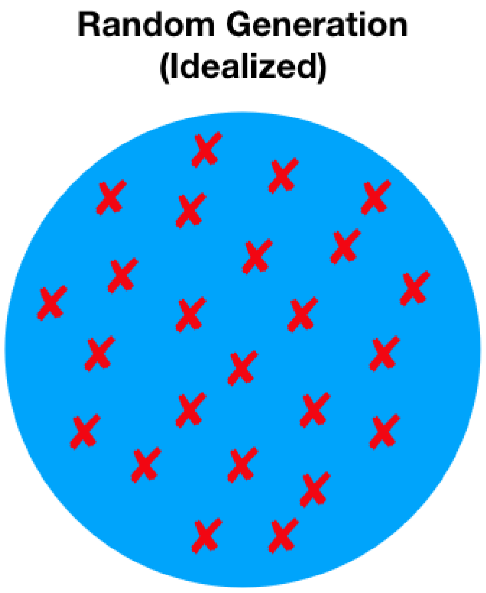
When performing automated test input generation, we are writing code that somehow explores this space of possible test inputs. While the details regarding input generation tend to be specific to the problem (e.g., for `min`, we are specifically generating two `int` values), there are certain general ways to perform this exploration. A relevant discussion follows in the next section.

2 Exploring the Space of Test Inputs

There are all sorts of ways in which we can explore the space of possible test inputs. This discussion focuses on three: random, bounded-exhaustive, and Swarm Testing.

2.1 Random Exploration

In an ideal world, random exploration will randomly select from the whole space of test inputs. This is visually represented below, where each red "X" above represents a selected test input.



2.1.1 Random Exploration Strengths

Theoretically, random exploration is great for getting a good breadth of testing. This means that random approaches should be able to cover a wide number of kinds of inputs, leading to better overall coverage of the system under test (SUT). Phrased another way, many parts of the SUT will be tested.

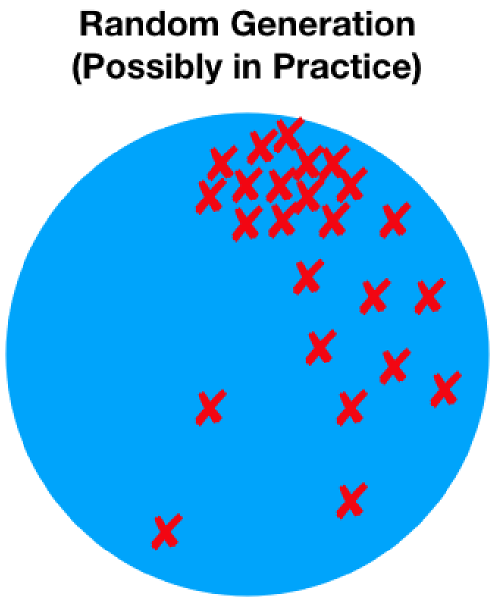
Random testing tends to be well-suited to situations where you're trying to maximize the number of bugs found. This is usually in an exploratory testing setting, where the testing itself is the main focus of your efforts. This sort of exploratory approach is used by Mozilla to test Firefox's JavaScript engine (using `jsfunfuzz` [9]), and by Microsoft to test the Z3 [2] constraint solver (using the work of Brummayer et al. [1]).

2.1.2 Random Exploration Weaknesses and Caveats

Some care needs to be taken with random testing. If the testing is *completely* random, where different tests are generated each time testing is requested, this can make development efforts difficult. For example, say you are working on a project wherein developers must pass 1,000 random tests before your code will be accepted. This can be nightmarish if you received a *different* set of 1,000 tests each time. Depending on your code, you might sometimes pass all the tests, whereas other times you may fail some tests. This is **bad** if random testing is a usual part of development, because it means your test suite is a moving target. Such tests are often called *flaky*, and they tend to get ignored since failure might happen sporatically.

That said, it's usually possible to set exactly how randomness is done. For example, if a fixed seed (`https://en.wikipedia.org/wiki/Random_seed`) is selected ahead of time, this will allow you to generate randomly **but** predictably. With fixed seeds, the same sequence of random numbers will always be chosen. In the example with the 1,000 tests, this means that the same 1,000 tests will always be generated, but that these tests will appear random. This can get the best of both worlds.

A major caveat with randomness is that we *might* not be generating a truly random sample, even though random numbers were somehow involved. To visually illustrate this, consider the image below:



As shown above, the sample of inputs generated is not well-distributed. One portion of the space is tested must more extensively than others, but another remains completely untested.

A simple example which illustrates this problem is shown here: `https://byorgey.wordpress.com/2013/04/25/random-binary-trees-with-a-size-limited-critical`

At that link, the author is interested in randomly generating binary trees. The algorithm chosen is pretty straightforward, and is illustrated below with C-like psuedocode:

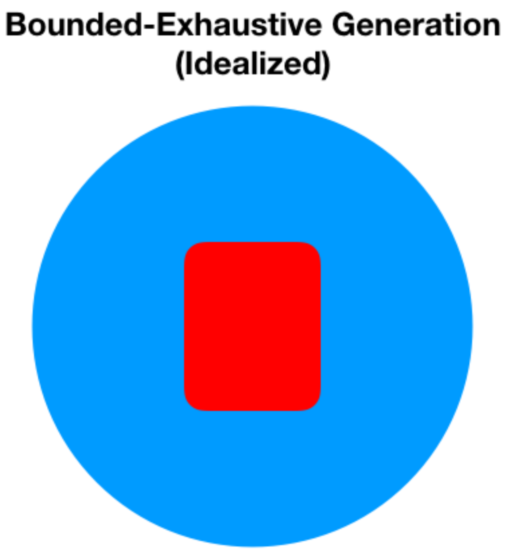
```
Tree makeTree() {
  r = randomNumberBetween(0.0, 1.0);
  if (r < 0.5) {
    return Leaf;
  } else {
    Tree left = makeTree();
    Tree right = makeTree();
    return InternalNode(left, right);
  }
}
```

While the above algorithm *does* involve randomness, it fails to generate a broad distribution of trees. Nearly half the trees generated consist of nothing but a single leaf (`Leaf`) node. This directly follows from the above algorithm: from the logic, with 50% probability, the first call to `makeTree` will result in a leaf.

Worse yet, the behavior after the first call to `makeTree` isn't exactly straightforward. Looking beyond the individual leaf nodes, the majority of generated trees contain fewer than 10 nodes total. However, this occasionally will generate trees with hundreds of nodes, and one generated tree (out of 100 generated) even contains over one hundred **thousand** nodes. The blog post discusses a way to fix this, though the mechanism described isn't very general. The point is that just throwing in random numbers does not guarantee a random distribution of test cases.

2.2 Bounded-Exhaustive Exploration

In contrast to random testing, *bounded-exhaustive* testing approaches will exhaustively generate every test input within some defined bounds. This is visually represented below:



The rounded red rectangle represents a bunch of test inputs which have been covered. The edges of this rectangle represent the selected bounds. Within these bounds, **every** test input is covered.

We saw an example of bounded-exhaustive testing when testing `min`. This is shown below for convenience, using Java:

```
public static final int MIN_BOUND = -100;
public static final int MAX_BOUND = 100;

@Test
public void testMin() {
  for (int small = MIN_BOUND; small < MAX_BOUND; small++) {
    for (int big = small + 1; big <= MAX_BOUND; big++) {
      assertEquals(small, min(small, big));
      assertEquals(small, min(big, small));
      assertEquals(small, min(small, small));
      assertEquals(big, min(big, big));
    }
  }
}
```

The above code testing `min` exhaustively explores values between `MIN_BOUND` and `MAX_BOUND`. Specifically, the first parameter to `min` can take on every value between `MIN_BOUND` and `MAX_BOUND - 1`. Even though this is a simple example, the values that the second parameter can take are already not very straightforward, as these depend both on the selected first parameter and `MAX_BOUND`.

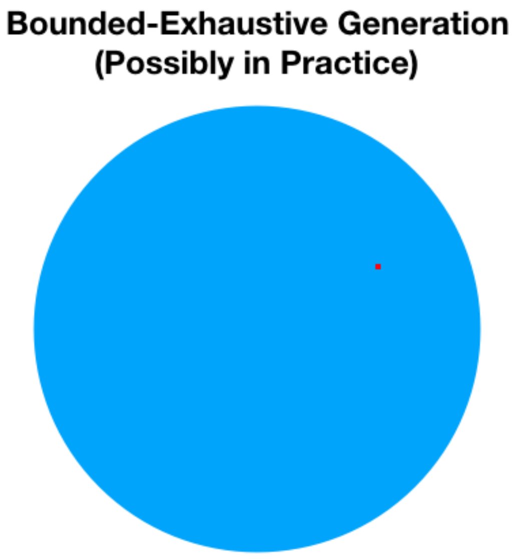
2.2.1 Bounded-Exhaustive Strengths

In theory, if the bounds are so large that they cover the entire space of inputs, bounded-exhaustive testing acts as a *verification* technique. That is, bounded-exhaustive testing can *prove* that code is correct, by showing that the code under test behaves correctly under every possible input. This illustrates the power of bounded-exhaustive testing at a theoretical level.

In practice, however, it's not usually possible to cover the entire space of inputs, so we must settle on some subset. The idea here is that, as long as the subset is still "big enough", this process will still mostly work as before. Phrased another way, this is the *small scope hypothesis*: most bugs can be exposed with rather small bounds, so testing with even relatively small bounds will still find most bugs.

2.2.2 Bounded-Exhaustive Weaknesses and Caveats

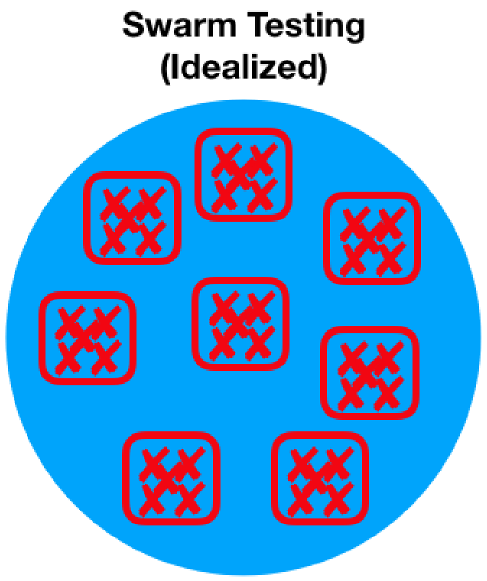
The working related to the small scope hypothesis is admittedly very murky: how big is "big enough", and what does "most bugs" mean? In practice, we don't usually know. Worse yet, given that most test input spaces are infinite, systematically testing with even trillions of inputs is the proverbial drop in the ocean. Visually, we might end up with the following situation:



As shown above, we cover only a *tiny* part of the input space (the red dot near the top right). Moreover, we're covering something off-center, which is intended to indicate that we're missing typical behavior. We're definitely covering everything in part of the space, but this part is so small and obscure that it might not matter in practice.

2.3 Swarm Testing

Swarm Testing [3] is based on the idea of focusing a random search on distinct subsets of the input space. Different subsets of the input space will be covered at different times. This is visually represented below:



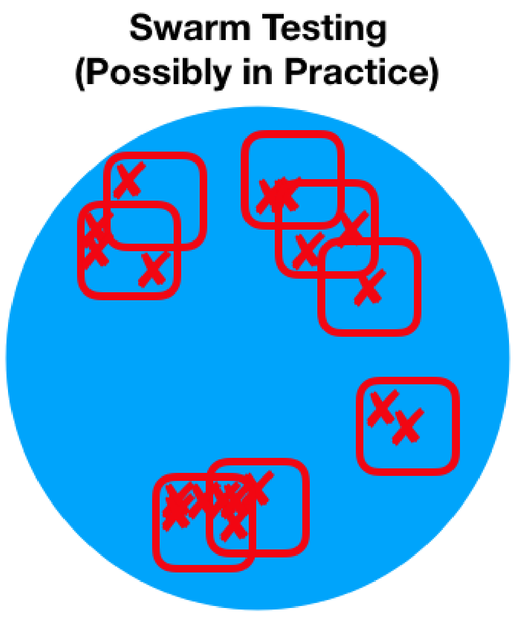
The red lines represent distinct subsets of the space, where red Xs represent individual tests. As shown, Swarm Testing in certain ways is taking the best of both random search and bounded-exhaustive search: the bounded-exhaustive part ensures that the whole space is covered, and the random part ensures a good sample of individual spaces.

2.3.1 Swarm Testing Background and Demonstration of Effectiveness

Swarm Testing was originally applied to CSmith [10], a tool which tests C compilers. Out of the box, CSmith uses a typical random search. Swarm Testing was implemented as something that simply cut out different C syntax rules (e.g., addition, pointer dereferencing, etc.) in order to focus-in on different subsets of C programs. This may seem counterintuitive: this intentionally **disables** the generation of C programs with selected features, meaning we can never possibly find bugs outside of those features. This, however, means that CSmith is forced to more heavily test the remaining enabled features, leading to bugs which only manifest when certain features are heavily used. The paper found that CSmith + Swarm Testing was more effective than CSmith out of the box, showing that Swarm Testing was more effective than plain random testing in this case.

2.3.2 Swarm Testing Weaknesses and Caveats

While the authors of Swarm Testing argue that it does not require significantly more complex implementation, this really depends on your test case generator. Moreover, we have been operating under the assumption that Swarm Testing sees the *best* of random and bounded-exhaustive generation. In theory, there is nothing stopping us from seeing the *worst* of random and bounded-exhaustive generation, as shown below:



As shown, the different subsets can overlap, either intentionally or otherwise. These subsets themselves might not be evenly distributed, leading to massive parts of the space being missed. With this in mind, care must be taken when dividing up the search space. Similarly, random testing might not be well-distributed within a subset, leading to parts of the space being skipped.

2.4 Which Approach Should I Use?

All of this discussion begs the question: which approach should I use? For better or for worse, the answer appears to be *it depends.* Random testing is pervasive in areas like compiler/interpreter testing (e.g., CSmith [10] for

C, `jsfunfuzz` [9] and LangFuzz [4] for JavaScript), with some evidence that the random approach is more effective than a bounded-exhaustive approach (e.g., `https://blog.regehr.org/archives/1246`). On an intuitive level, the need for randomness in testing compilers and interpreters makes sense; as inputs, programs take on many dimensions, making it completely impractical to test much of anything with a bounded-exhaustive apporach. However, I'm unaware of a full-scale study showing that randomness is the *right* choice here; much of this may be entrenchment of ideas.

Bounded-exhaustive testing intuitively seems well-suited to problems with relatively few dimensions, where it is feasible to cover a decent amount of each of them. Some studies argue that the small-scope hypothesis is true [8], and more widely that bounded-exhaustive testing is superior [5, 7]. Even so, this hardly seems like a general conclusion, and some of these studies aren't peer-reviewed.

Swarm Testing [3] seems to capture nice elements of random and bounded-exhaustive testing, and has been demonstrated to be effective in other studies [6]. However, it's still personally unclear to me if this is *always* the case.

Moreover, these approaches are not mutually exclusive. It's entirely possible to use one approach on one test input dimension, and another approach on another dimension. In practice, people often pick one apporach for all dimensions, but there is no fundamental reason why this has to be the case.

Overall, there is not one clear winner. In practice, people usually pick one and stick with it for all dimensions, with no experimentation. I suspect that this is because changing between different approaches can be difficult from an implementation standpoint, and can lead to subtle gotchas (see, for example, `https://blog.regehr.org/archives/1246`). I similarly suspect that this is because our tools to measure testing effectiveness (e.g., coverage and mutation analysis) are pretty poor, so determining which is best isn't necessarily straightforward. If the results of code coverage and mutation analysis look good, you're probably using at least an ok approach.

References

[1] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging smt solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, pages 1–5, New York, NY, USA, 2009. ACM.

[2] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 78–88, New York, NY, USA, 2012. ACM.

[4] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[5] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. Reducing the costs of bounded-exhaustive testing. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 171–185, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 65–76, New York, NY, USA, 2015. ACM.

[7] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An evaluation of exhaustive testing for data structures. Technical report, MIT Computer Science and Artificial Intelligence Laboratory Report MIT -LCS-TR-921, 2003.

[8] Johannes Oetsch, Michael Prischink, Jörg Pührer, Martin Schwengerer, and Hans Tompits. On the small-scope hypothesis for testing answer-set programs. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*, KR'12, pages 43–53. AAAI Press, 2012.

[9] Jesse Ruderman. Introducing jsfunfuzz, 2007.

[10] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.