Search-Based Test Data Generation

CS454 AI-Based Software Engineering

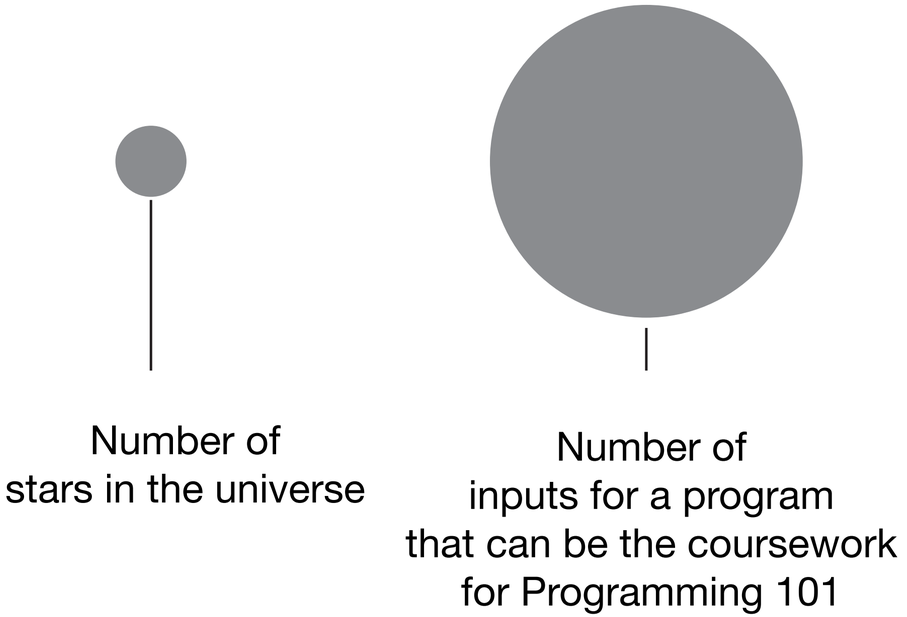
Software Testing

- A form of dynamic verification, i.e., you check the correctness of implementation by actually executing the program and checking the output.
- Problem: unless you execute all possible inputs, there is no way of guaranteeing that the program is correct, period.
  - Executing all possible inputs is typically impossible, just because there are too many.
  - This led to the (in)famous quote from Dijkstra — “Testing can only prove the presence of bugs, not their absence” — which is entirely correct.

Software Testing

The case of triangle program

- Triangle program takes three integers and tells you the type of triangle that has three sides as long as the numbers.
- 32bit integers: between -2^{31} and $2^{31}-1$, there are 4,294,967,295 numbers
- The program takes three: all possible combination is close to 8^{28}.
- Approximated number of stars in the known universe is 10^{24}.



A Question Follows: When can I stop testing?

- All testing techniques are essentially a triplet:
  - Sampling of Input: since exhaustive testing is impossible, what is my subset?
  - Test Oracle: if I execute the sampled input, how do I check whether the output is as expected?
  - Adequacy: when do I know that my testing is adequate (i.e., sufficient)?
- For example, a basic fuzzing technique is *(random sampling, checking for crashes, until the time runs out)*.

Structural Testing

- A test adequacy criterion that aims to execute all parts of program at least once (i.e., achieve coverage): for example, when your testing executes all statements, you achieve 100% statement coverage.
- Remember that *this does not guarantee anything*: **coverage is necessary, but not sufficient, condition for detecting faults**.
  - Executing `x = a / b` only means something when `b = 0`.
- This gives us a stopping point: technically one can stop testing when 100% coverage is reached.
- This also allows us to compare two inputs: the one that achieves higher coverage has higher chance of detecting a bug, given that all other things are equal.

Test Data Generation for Structural Testing

How do I automatically cover arbitrary parts of my program?

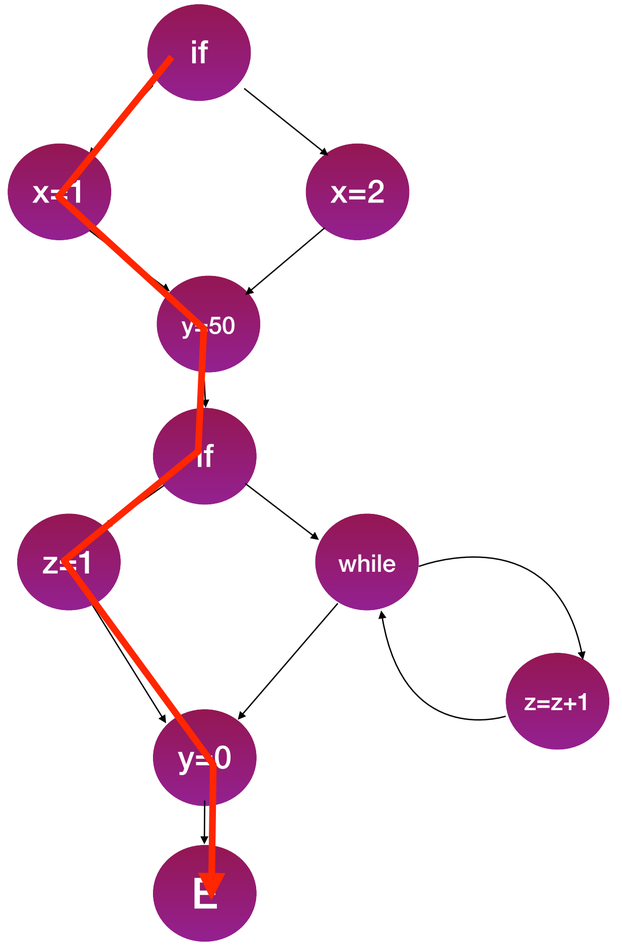
- A mature research area with many widely-studied approaches.
- Traditional approaches:
  - You. Think and write down.
  - Random: generate random inputs until you cover everything (not very likely in some cases)
  - User session: but they weren’t testing really…
- How do we formulate this as optimisation?

Path Condition

- A collection of predicates that leads the program execution down to a specific path:

```
if(y > 13):
    x=1
else:
    x=2
y = 50
if(w == 4):
    z = 1
else:
    while some_condition():
        z = z + 1
y = 0
```

What is the path condition?



y > 13 && w == 4

Predicate to Distance

- Predicates are *discrete* Boolean logic, i.e., they are either true or false. Optimisation is about maximising or minimising a real, *continuous*, function.
- Can we define “how far until this becomes true”?
- For example, if you want to satisfy the predicate `x == y`, you convert this to distance of `b = |x - y|` and seek the values of `x` and `y` such that minimise `b` to 0. Hey, `x` is now identical to `y`.
- Similarly, to satisfy the predicate `y >= x`, convert this to `b = x - y + K` and seek the values of `x` and `y` such that minimise `b` to 0. Now `y` that is larger than `x` by `K`.

The Branch Distance Recipe

Predicate	f	minimise until..
a > b	b - a + K	f < 0
a >= b	b - a + K	f <= 0
a < b	a - b + K	f < 0
a <= b	a - b + K	f <= 0
a == b	|a - b|	f == 0
a != b	-|a - b|	f < 0

B. Korel, “Automated software test data generation,” IEEE Trans. Softw. Eng., vol. 16, pp. 870–879, August 1990

How to reflect the concrete execution?

```
def foo(x):
    if x == 42:
        print("Hey!")
    ...
    return
```

This is almost trivial. We can directly optimise the input argument to achieve branch coverage.

```
def foo(x):
    y = complex_computation(x)
    if bar(y) == 42:
        print("Hey!")
    ...
    return
```

Unless we know how to inverse arbitrary functions, this is much harder to optimise `x` for.

Idea: we measure distance not directly in terms of input arguments, but concretely at each predicate.

How to overcome concrete control flow?

That is, you cannot measure anything from predicates you are not executing.

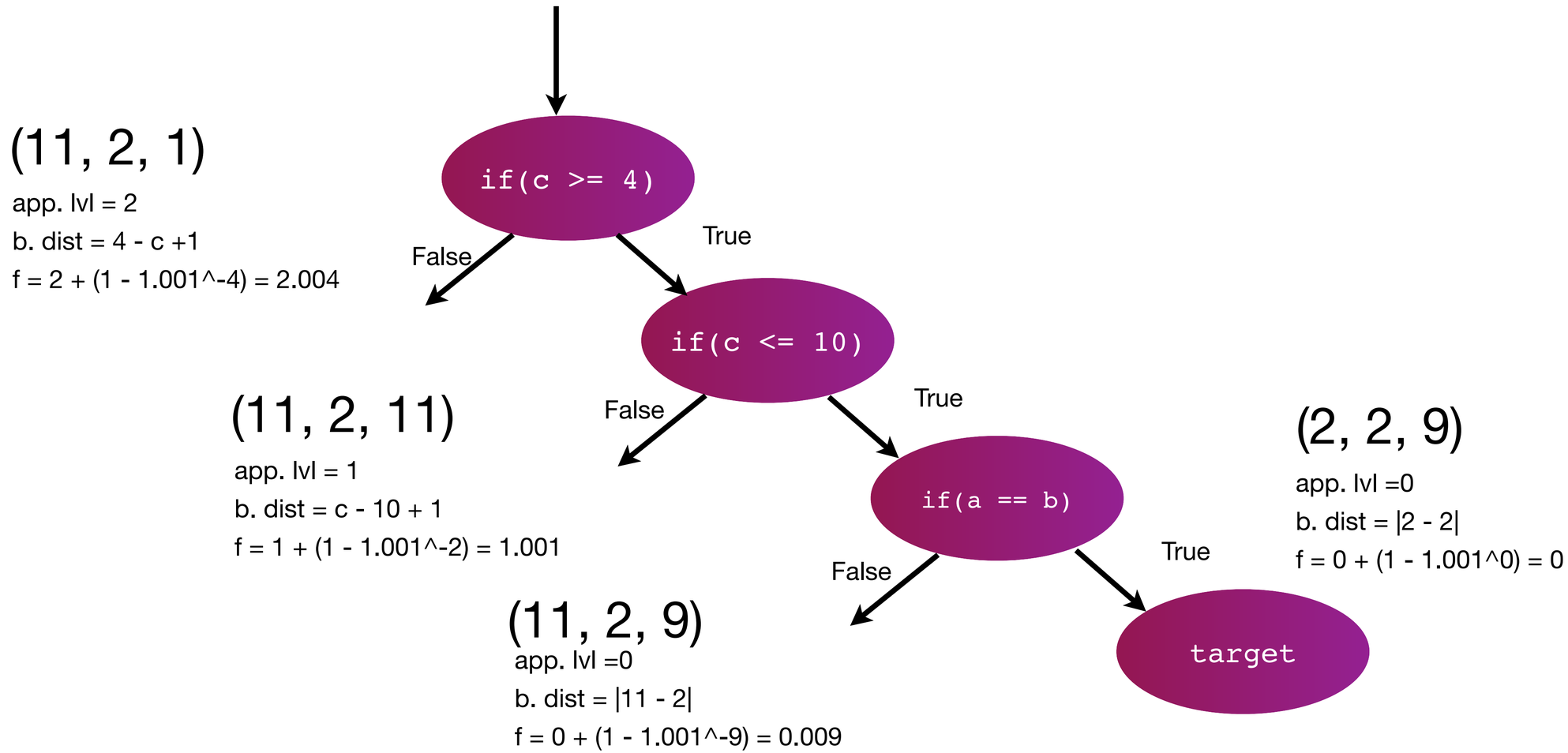
```
def testme(x, y):
    if foo(z) == 42:
        print(...)
        z = complex_computation(x)
        if y == bar(z):
            print("Here!") // Target
    ...
    return
```

Idea: we have to gradually drive the execution to our target.

Fitness Function for Structural Testing

- Fitness function for branch coverage = [approach level] + normalise([branch distance])
- For a target branch and a given path that does not cover the target:
  - Approach Level: number of un-penetrated nesting levels surrounding the target
  - Branch Distance: how close the input came to satisfying the condition of the last predicate that went wrong
  - Normalisation: so that we can add count (approach level) to distance — otherwise branch distance will dominate the fitness. A popular choice is $norm(b) = 1 - 1.001^{-b}$.

Test input (a, b, c), K = 1



What do we actually need?

1) Branch Distance Instrumentation

- After instrumentation, all branch predicates should:
  - preserve the original semantic, i.e., should return a boolean value based on the original predicate
  - record the branch distance somewhere, so that the fitness can be calculated
- How? :)
  - Rewrite, using side-effects.

What do we actually need?

2) Approach Level Computation

- There are two paths:
  - One that we need to follow if we want to reach the target predicate.
  - Another that the current execution follows.
  - We need to identify the predicate from which two paths diverge, and count the predicates between the divergent point and the target.
- How?
  - Control Dependent Graph.

Search-Based Test Input Generation

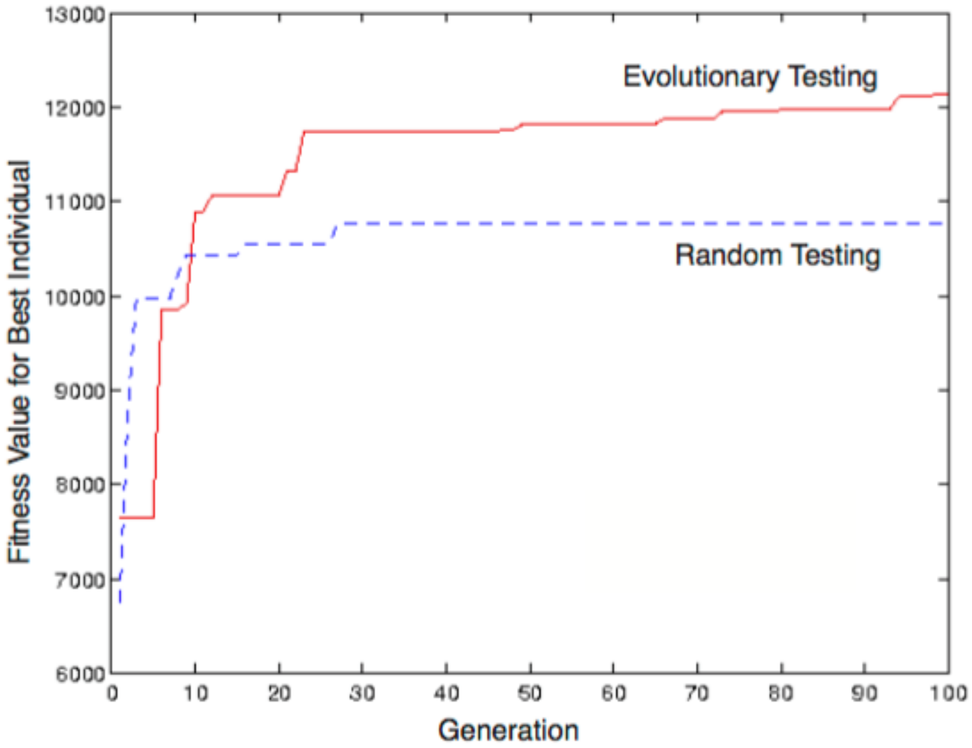
- Once you have an instrumented target program (with branch distance and approach level computation), you can plug in any meta-heuristic optimisation and start minimising the fitness function!
  - Hill Climbing works well if there exists a nice gradient towards the input values that achieve the coverage (e.g., imagine a program whose branches are all numerical comparisons).
  - Genetic Algorithm works well if the input value that achieves the coverage can be assembled from many smaller parts (e.g., imagine a program that checks a valid ISBN number, or a mobile phone number).

Weaknesses

- Distance for non-numerical predicates: How about strings? Arbitrary data structures?
- Generating random instances as well as neighbours of complex data types: What is a random tree? A random Python source code? A random PDF?

Strengths

- Anything that 1) *depends on the input* and 2) *can be captured in an objective function*, can be optimised for.
- For example, non-functional properties like execution time. GA found inputs that result in longer execution time in real time systems, much better than random.



J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing.

Real-Time Systems, 15(3):275 – 298, 1998.

Strengths

Non-functional Properties to Optimise for

- Energy Consumption: what is the GUI event sequence that results in wasted power consumption in Android apps? (https://ieeexplore.ieee.org/abstract/document/8812097)
- Memory Usage: what is the input that maximises the memory usage of the target program? (https://dl.acm.org/doi/10.1145/1276958.1277175)

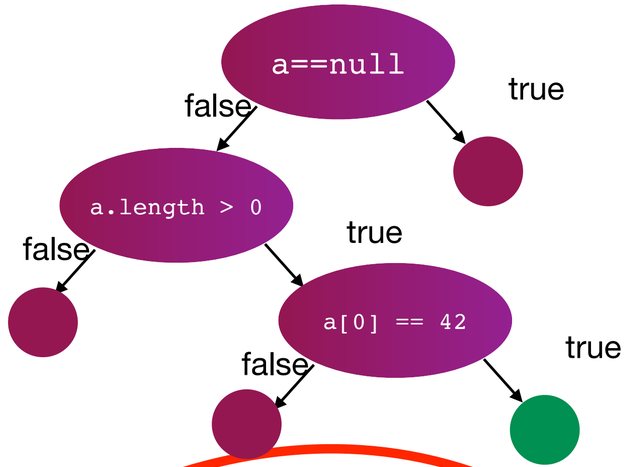
Alternatives: Fuzzing

- Essentially a (more) random search + a very lightweight instrumentation (e.g., just coverage)
  - Random search: start with a seed input chosen from a pool, then apply a random mutation
  - Instrumentation (grey-box fuzzing): just check whether a newly mutated input achieved any new coverage — if so, retain as a seed.
- Less rigorous search, therefore takes much longer to do certain things.
- Much more lightweight instrumentation, therefore easier to apply to larger systems!

Alternatives: Symbolic Execution

- Think of it as how you execute a code in your head, only using a program
- Start with a basic (random) input, execute program and collect observed predicate conditions from the concrete execution
- Negate the last clause of the observed condition
- Solve the resulting constraint using a solver (thereby covering one more branch, i.e., the other branch from the last predicate)
- Repeat!

```
void testme(int[] a)
{
   if(a == null) return;
   if(a.length > 0)
   {
      if(a[0] == 42)
         throw new Exception("bug");
   }
}
```



Solve

Execute

Constraints to Solve	Data	Observed Path Condition
	null	a==null
a!=null	{}	a!=null && !(a.length > 0)
a!=null && a.length > 0	{0}	a!=null && a.length > 0 && a[0] != 42
a!=null && a.length > 0 && a[0] == 42	{42}	No more path!

Negate last condition and choose another path

Alternatives: Dynamic Symbolic Execution
(aka Concolic Testing)

- This is a very powerful idea, and in many cases much more efficient than search-based approaches
  - As long as the SMT solver is capable of solving the negated condition, we are guaranteed to cover one more branch!
  - There are long-standing challenges in SMT solvers (e.g., handling floating point numbers, especially divisions, etc), but we have seen many advances.
- Weakness: anything that cannot be captured symbolically cannot be solved for (e.g., non-functional properties)

Alternatives: LLMs and Agents

- We can certainly prompt LLMs to generate test inputs 😎
- They will likely generate inputs for the happy path, i.e., the most representative, friction-less execution path, first. Subsequently, it will take various prompting strategies to cover remaining branches. However, there are a couple of obvious limitations:
  - It can only manage “state” via the context, which is much noisier than simple algorithmic loops. LLMs may have a hard time covering “additional” branches.
  - The guidance towards specific branches is very weak, as we can only “instruct/prompt”.

The Oracle Problem

- Oracle: any mechanism that checks whether the observed output for a given input is correct.
  - For example: `assertTrue(test_me(“the answer to life the universe and everything”) == 42)`
- The oracle problem means that a fully automated oracle for any arbitrary input is not feasible.
  - Suppose you can determine what the correct output is for any arbitrary input —> Such mechanism can be used to actually write THE CORRECT PROGRAM…?

Different Types of Oracles

(Other than you writing a 1-to-1 input-output mapping, that is)

- Regression Testing: your aim is not to check correctness, but to ensure that the program behaviour do not change from the previous version —> your oracle is the comparison to the output from the previous version.
- Differential Testing: similar to regression testing, but you are porting to a different environment —> your oracle is the comparison to the original version.
- Implicit Oracle: there are conditions that no program should violate, like “do not crash” or “do terminate”. While the actual checking mechanism may not always be obvious, it is clear what they mean :)

Again, LLMs and Agents?

- We can ask LLMs and Agents to generate oracles, too 😎
  - And they will have similar limitations, too. They will likely be capable of figuring out the expected outcomes from the happy path, but may struggle for other, rarer executions.
  - “Quis custodiet ipsos custodes?” (“Who watches the watchmen?”)

Summary

- Testing is essentially a (very lopsided) sampling game - you are either lucky enough to find an input that will trigger a bug, or now.
- Any input generation technique is an attempt to balance cost and diversity.
- Meta-heuristic search and symbolic execution are two foundational approaches towards structural test input generation.
- LLMs and agents will likely cover many of the common cases, but there are fundamental differences and some limitations to keep in mind.